THE UNIVERSITY OF ALBERTA

SYNTAX DIRECTED ANALYSIS

by

Thomas P. McIntosh

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

December, 1967

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled SYNTAX DIRECTED ANALYSIS submitted by Thomas P. McIntosh in partial fulfilment of the requirements for the degree of Master of Science.

ABSTRACT

This thesis reviews various techniques for determining the meaning of statements written in a programming language using the syntax specifications of the language. The review covers the basic theory of phrase structure grammars, syntax specifications, and syntax directed analyzers. Working models of the three main syntax directed analyzers, conventional, multiple parse, and transition diagrams were constructed and tested using APL.

ACKNOWLEDGEMENTS

I express my appreciation to Professor W.S. Adams for the guidance given me in the preparation of this thesis and to Professor D.B. Scott, Head of the Department of Computing Science for providing computing facilities and financial assistance while this research was being done.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Communication and Translation

Whenever a person wishes to communicate with another it is necessary for him to send a message which the receiver can understand. If the people have a common language this process is simple. If their languages are different a translation must be made.

The translation process is direct if there is a one-to-one correspondence of the words in the two languages. However, a word usually has a number of meanings in one language. These may be represented by a set of words in the other language. In order for the translator to decide which meaning the sender intended it is necessary to consider the word in relation to surrounding words, i.e. determine the meaning of the word by information supplied by its context.

The same general concept of communication applies to computers and computer programs. The programmer plays the role of the sender. His programming language is a combination of natural language and mathematical notation. The computer is the receiver. Its language is a numerical representation of the instructions it can perform. If a

programmer has a problem which can be described in a programming language it can only be run on a computer if a translation is made.

Early computers required that this translation be done by the programmer. Later, with the development of larger and faster computers, it became feasible to assign the translation process to the computer itself. Iverson (1962) places such translators in the following categories, compilers, assemblers, generators, and interpreters. Compilers accept a program expressed in an argument or source language and reproduces the program in a function language, usually machine code, to be run later. Assemblers are special compilers in which the statements of the program are virtually independent of each other. Thus, the statements can be considered one at a time and are simple (not compound) so there are no contextual meanings. Assemblers generally are used to translate so-called "symbolic" programs into machine code. A generator produces any one of a set of function programs based on a parameter which is supplied to it. Generators are frequently incorporated in compilers. An interpreter executes the segment of a function program corresponding to a statement of the argument program immediately after it is produced. The statements of the argument program are selected in a sequence determined by the function program.

The conventional approach to translator writing requires a separate translator for each computer-programming language combination desired. This method of producing a translator is expensive and time consuming as the rules of the language are embedded in the structure of the translator program itself. The proliferation of computers and languages had made this procedure uneconomical and various alternatives have been suggested.

1.2 Linguistics: A New Approach in Translators

In 1960 a new approach to compiler writing began to evolve. Linguistic theory had developed the principle of phrase structure grammars as a tool for the study of the structure of natural languages. Although compiler writers had indirectly used the structure of a language in their translators, linguistic theory now provided the basis for an organized use of structure in translators. (Metcalfe (1964), Davis (1966))

The production of a programming language system for a computer requires a union of the definition of the language, the design of the translator, and the characteristics of the computer. Prior to 1960 these factors were combined in the development of the translator. Syntax oriented translators use aspects of linguistic theory to separate the definition of the language and the design of the translator.

The first separate definition of a programming language took place with the introduction of ALGOL. In 1961 Irons demonstrated the feasibility of syntax oriented translators by producing a compiler which worked from a set of specifications for a language. Since then various means of achieving this separation have been developed. (Irons (1961), Davis (1966))

Although syntax oriented techniques are developing rapidly in a number of directions, all the methods stem from phrase structure grammars which comprise only one facet of linguistics. This thesis explains the important aspects of phrase structure grammars and considers topics related to the use of such grammars in translation systems. In particular, it will review certain syntax oriented techniques for recognizing structures of a program based on specifications of the syntax of the language involved.

The study of recognition algorithms consists mainly of working models in a programming language called APL. This language is an automated version of a notation which was first described in Iverson's 'A Programming Language' (1962).

APL was chosen because its concise notation enables one to describe a system in detail and at the same time provides an operating model which can be run on a computer. In addition, APL is used in a time-sharing environment, which permits easy modifications to models and an immediate determination of the effects.

CHAPTER II

LANGUAGE AND MEANING

2.1 Linguistic Definitions and Programming Languages

Syntax analysis of programming languages is a result of the similarity between programming languages and natural languages. Because programming language analysis has borrowed many terms from natural language analysis, it is advantageous to consider the main terms that have developed in linguistic theory.

A written _language_ conveys meaning by means of objects or marks which are catenated to form strings. The _syntax_ of a language refers to the linear arrangement of these objects. A _rule of syntax_ states some permissible (or prohibited) relation between objects. The _grammar_ of the language is the set of syntactic rules. _Semantics_ defines the relationship between an object and the set of meanings attributed to the object. A _symbol_ is an object to which at least one meaning has been attributed. _Pragmatics_ defines the relation between a symbol and its user. An object must have at least one meaning to be of value to a user. A _rule of pragmatics_ is applied by a user to select from the set of meanings attributed to a symbol, that particular meaning which is significant to a particular user at a particular time. A _sentence_ is the smallest unit which a meaningful

string can form in natural language. (Ingerman (1966)) The parse of a sentence indicates what rules of the grammar were used to form the sentence.

Natural languages are more or less capable of describing the wide range of topics encountered by humans. Furthermore, much meaning is often contained in context within a sentence. Programming languages on the other hand need only describe the limited number of operations which can be performed by a computer and the operands which are used. The operands are the names of memory locations, registers, or external devices. The linguistic terms can now be defined more formally and simply.

Language now becomes a method of describing a process through the use of symbols which represent operations and operands. Syntax is concerned with the arrangement of symbols, independent of their meaning. Semantics, which relates symbols and their meanings, is restricted in that each symbol has only a small number of possible well-defined meanings. Pragmatics is concerned with how the translator will select the meaning of a symbol in the source language. The smallest unit for a meaningful string in a programming language is called a <u>statement</u> rather than a sentence. (Gorn (1961))

2.2 Structure and Meaning

The translation of a string of symbols from one language to another consists of preparing a string of symbols in the second language which has the same meaning as the original. Attempts at having computers translate natural languages have not been satisfactory because of the wide variation in permitted structures and the extensive use of contextual meaning. In order to assign the translation of programming languages to computers it was necessary to define the structure of statements which could be used to convey a given meaning. Thus, determining the structure of a statement is equivalent to determining the meaning of the statement.

The meaning of a statement in a programming language is absolute or deterministic in that it can be uniquely explained in terms of changes which are effected on a certain set of variables by obeying the statement. For example, execution of the statement $A \leftarrow B+C$ will result in the current value of the variable A being replaced by the sum of the values of variables B and C. Since machine language can describe all basic operations on variables, the translation process can be well defined. (Wirth and Weber (1966))

In order to determine the meaning of statements written in a programming language, three topics must be considered. First the restrictions and rules which define the structures in the language must be presented. Second these rules must

organized so they can be used by a translator. Finally, algorithms must be designed and developed for the parsing of statements so as to determine their structure in relation to the rules.

CHAPTER III

PROGRAMMING LANGUAGE GRAMMARS

3.1 Phrase Structure Grammars

Formal grammars, which define languages suitable for automatic translation, can be classified by the restrictions placed on the syntactic structures in the language. One classification is: phrase structure grammars, standard form grammars, bounded context grammars, operator grammars and precedence grammars. Of these, phrase structure grammars are the most general and will be defined and described in detail. The remaining grammars will then be discussed.

The following account of phrase structure programming languages is based on the definitions given by Wirth and Weber (1966).

A vocabulary $\underline{V}$ is a set of symbols denoted by capital Latin letters S, T, U, etc. Finite sequences of symbols - including the empty sequence $\underline{N}$ - are called strings and are denoted by small Latin letters x, y, z, etc. The set of all strings over $\underline{V}$ is denoted by $\underline{V}^*$ and $\underline{V} \subseteq \underline{V}^*$.

A simple phrase structure system is an ordered pair $(\underline{V}, \underline{R})$ where $\underline{V}$ is a vocabulary and $\underline{R}$ is a finite set of syntactic rules or productions $\underline{r}$ of the form $U \rightarrow x$ where $x \neq U$, $U \in \underline{V}$, and $x \in \underline{V}^*$. For $\underline{r} \equiv U \rightarrow x$, U is called the

left part and x is the right part of $\underline{r}$. The component U is called the metaresult and the components of x are called the metacomponents.

The string y directly produces z $(y \overset{\cdot}{\to} z)$ and conversely z directly reduces into y, if and only if there exist strings u, v such that $y = uUv$ and $z = uxv$ and the rule $U \to x$ is an element of $\underline{R}$. y produces z $(y \overset{*}{\to} z)$ and conversely z reduces into y if and only if there exists a sequence of strings $x_o, \ldots, x_n$ such that $y = x_o$, $x_n = z$ and

$$x_{i-1} \overset{\cdot}{\to} x_i \qquad (i=1,\ldots,n;\ n\geq 1).$$

In this case z is a derivation of y.

A simple phrase structure grammar is an ordered quadruple $\underline{G} = (\underline{V}, \underline{R}, \underline{B}, \underline{A})$. $\underline{V}$ and $\underline{R}$ form a phrase structure system and $\underline{B}$ is a subset of $\underline{V}$ such that none of the elements of $\underline{B}$ (called basic or terminal symbols) occurs as the left part of any rule of $\underline{R}$. All elements of $\underline{V} - \underline{B}$ (called non-terminal symbols) occur as the left part of at least one rule. $\underline{A}$ is the symbol which occurs in no right part of any rule of $\underline{R}$ and is referred to as the head of the language.

The letter U (or U_i) will denote a non-terminal symbol, i.e. $U_i \in \underline{V} - \underline{B}$.

x is a sentence of $\underline{G}$ if $x \in \underline{B}^*$, i.e. x is a string of basic symbols, and $\underline{A} \overset{*}{\to} x$.

A simple phrase structure language $\underline{L}$ is the set of all strings x which can be produced by $(\underline{V}, \underline{R})$ from $\underline{A}$:

$$\underline{L}(\underline{G}) = \{x \mid \underline{A} \overset{*}{\to} x \wedge x \in \underline{B}^*\} .$$

Let $U \overset{*}{\to} z$. A parse of the string z into the symbol U is a sequence of syntactic rules $\underline{r}_1, \underline{r}_2, \ldots, \underline{r}_n$ such that $\underline{r}_j$ directly reduces z_{j-1} into z_j $(j=1,\ldots,n)$ and $z = z_o$, $z_n = U$. A canonical parse is a parse which proceeds strictly from left to right in a sentence and reduces a left-most part of a sentence as far as possible before proceeding further to the right.

If $z_k = U_1 U_2 \ldots U_m$ (for some $1 < k < n$), then z_i $(i < k)$ must be of the form $z_i = u_1 u_2 \ldots u_m$, where for each $s = 1,\ldots,m$ either $U_s \overset{*}{\to} u_s$ or $U_s = u_s$. The canonical form of the section of the parse reducing z_i into z_k shall be $\underline{r}_1, \underline{r}_2, \ldots, \underline{r}_m$ where the sequence $\{\underline{r}_s\}$ is the canonical form of the section of the parse reducing U_i into U_s. Clearly $\{\underline{r}_s\}$ is empty if $U_s = u_s$, and is canonical if it consists of one element only.

An unambiguous syntax is a phrase structure syntax with the property that for every string $x \in \underline{L}(\underline{G})$, there exists exactly one canonical parse.

An environment $\underline{E}$ is a set of variables whose values define the meaning of a sentence. An interpretation rule defines an action (or a sequence of actions) involving a subset of the environment.

A phrase structure programming language $\underline{L}_p$ ($\underline{G}$, $\underline{I}$, $\underline{E}$) is a phrase structure language $\underline{L}(\underline{G})$ where $\underline{G}$ ($\underline{V}$, $\underline{R}$, $\underline{B}$, $\underline{A}$) is a phrase structure syntax, $\underline{I}$ is a set of (possibly empty) interpretation rules such that a unique one-to-one mapping exists between elements of $\underline{I}$ and $\underline{R}$, and $\underline{E}$ is an environment used by the elements of $\underline{I}$.

The meaning $\underline{M}$ of a statement $x \in \underline{L}_p$ is the effect of the execution of the sequence of interpretation rules $\underline{I}_1$, $\underline{I}_2,\ldots,\underline{I}_n$ on the environment $\underline{E}$, where $\underline{r}_1$, $\underline{r}_2 \ldots \underline{r}_n$ is a canonical parse of the sentence x into the symbol $\underline{A}$ and $\underline{I}_k$ corresponds to $\underline{r}_k$ for all k. The meaning may have the effect of changing values of variables or of changing the environment by introducing or removing variables.

Phrase structure grammars were first introduced and studied by Chomsky as devices for generating sentences in natural languages. By imposing more and more severe restrictions on the productions, four types of grammars were defined.

Type 0 grammars place no restrictions on the forms of the productions.

Type 1 grammars (context dependent) have productions of the form $x \rightarrow y$ where

$$x \equiv e\ U\ f, \quad y \equiv e\ w\ f, \quad \text{and} \quad w \neq \underline{N}.$$

Type 2 grammars (context free) have productions of the form $x \rightarrow y$ where

$$x \equiv e\ U\ f, \quad y \equiv e\ w\ f, \quad w \neq N, \quad \text{but} \quad U \rightarrow w.$$

Type 3 grammars (finite state) are of the form $x \rightarrow y$ where

$$x \equiv e\ U\ f, \quad Y \equiv e\ w\ f, \quad w \neq \underline{N}, \quad U \rightarrow w \quad \text{but}$$

all productions are of the form

$$w \equiv t\ U_1 \quad \text{or} \quad w \equiv t, \quad U_1 \text{ is a non-terminal}$$

and t is a terminal. (Chomsky (1959), Landweber (1964))

Subsequent sections will consider the relationships between Type 2 grammars and programming languages.

3.2 Standard Form Grammars

In order to work efficiently, and in some cases at all, some algorithms for the syntactic analysis of phrase structure languages prohibit infinite left recursive productions. A non-terminal $U \in \underline{V} - \underline{B}$ is left recursive if there exists a production rule $U \overset{*}{\rightarrow} U\ x$ for some $x \neq \underline{N}$. Left recursion can be removed by transforming the grammar to standard form in which all of the rules of $\underline{R}$ are of the form:

$$U \rightarrow T \quad \text{or}$$

$$U \rightarrow T\, U_1\, U_2 \ldots U_n,\ n \geq 1, \quad \text{where}$$

$$T \in \underline{B} \quad \text{and} \quad U_i \in \underline{V} - \underline{B}.$$

Standard form grammars can have no infinite left-going structures. (Greibach (1965), Galler and Perlis (1967))

Kunos' article (1966) describes a proof due to Greibach which shows that for a given context free grammar $\underline{G}$, a standard form grammar $\underline{G}_s$ can be constructed which generates the same language as generated by $\underline{G}$. However, when $\underline{G}_s$ is used to parse a string, it does not produce the same structural descriptions as $\underline{G}$. The article also contains an algorithm designed by Abbot which converts a given context free grammar into an augmented standard form grammar supplemented by additional rules describing its derivation from the original context free grammar. It is then possible to correct the structural descriptions supplied by the standardized grammar.

Kurki - Suonio (1966) has shown that it is not necessary to transform the grammar to standard form if removal of left recursion is sufficient. His system entails defining new non-terminal symbols which are used to modify the current rules.

3.3 Bounded Context, Operator, and Precedence Grammars

Bounded context grammars, a subset of type 2 phrase structure grammars, are grammars which are restricted so that the structure of a substring of a sentence may be determined by

considering a limited portion of the ~~substring~~. For any specified bound on the number of contextual characters considered, it is possible to determine if the grammar is bounded. Bounded context grammars are free from syntactic ambiguity and can form models for most languages used in computer programming. (Floyd (1964a))

In an effort to design an efficient syntax oriented compiler, Floyd (1963) developed the concepts of operator and precedence grammars. These grammars are subsets of bounded context grammars.

If no production of a phrase structure grammar $\underline{P}$ takes the form

$$U \rightarrow x\ U_1\ U_2\ y,$$

where U_1, U_2 are nonterminals, then $\underline{P}$ is an operator grammar and $\underline{L}_p$ is an operator language. In an operator grammar, there are three possible relations (denoted by $\doteq$, $\gtrdot$ and $\lessdot$), which two terminal characters T_1 and T_2 may take. The relations are defined as follows:

1. $T_1 \doteq T_2$ if there is a production $U \rightarrow x\ T_1\ T_2\ y$ or $U \rightarrow x\ T_1\ U\ T_2\ y$.
2. $T_1 \gtrdot T_2$ if there is a production $U \rightarrow x\ U_1\ T_2\ y$ and a derivation $U_1 \overset{*}{\rightarrow} z$ where T_1 is the rightmost character of z.

3. $T_1 \lessdot T_2$ if there is a production $U \rightarrow x\ T_1\ U_1\ y$ and a derivation $U_1 \overset{*}{\rightarrow} z$ where T_2 is the left-most terminal character of z.

One, two, or all of the above relations may hold.

A precedence grammar is an operator grammar for which no more than one of the above three relations holds between any ordered pair T_1, T_2 of terminal symbols. The relations are then called precedence relations. The precedence grammars form models of mathematical and algorithmic languages which may be anlayzed mechanically by a simple procedure based on a matrix representation of the precedence relations between characters.

Wirth and Weber (1966) point out that precedence grammars are unambiguous in the sense that the sequence of syntactic reductions applied to a sentence is unique for every sentence in the language. Since every sentence is uniquely analyzed and each rule has an interpretation rule, the definition of meaning is exhaustive. Thus, every sentence has one and only one meaning, a necessity for programming languages. The authors have also developed an algorithm which decides whether a given grammar is a precedence grammar, and if so performs the desired transformation into data representing the reductive form of the grammar. This reductive form can be used to illustrate how a statement may be reduced to the head of the grammar.

3.4 Structural Connectedness

The theory of phrase structure grammars was developed as a means of studying techniques to generate and/or recognize sentences in natural languages. Irons (1964) felt that the Type System developed by Chomsky was too broad to be useful for classifying the analysis algorithms for various grammars. In its place he suggested the concept of "structural connection" as a means of classifying various languages.

The classification is based on the complexity of the interaction between parses on disjoint substrings of a parsed string. The lowest level of the classification is structurally unconnected. The next level, structurally connected, describes systems in which the symbols surrounding a string determine its parse. Finally, structurally connected in depth refers to a system in which the parse of one string depends on parses of other strings.

The analyzers for various grammars can be classified according to the class of grammar which they will process. A recognizer for a structurally unconnected system is basically a table-lookup algorithm. Recognizers for structurally connected systems are efficient for a limited number of symbols to the left or right. However, if several substructures are present, tentative analysis may be required. Systems which are structurally connected in depth to the left may require a dynamic modification of the grammar specifications

or a complicated intermediate tabling procedure for left-to-right recognizers. Recognizers for systems which are structurally connected in depth to the right may be multiple pass or may be non-existent.

Symbolic languages which require assemblers to produce machine code are structurally unconnected. The grammars described in this paper are termed structurally connected. High level languages are structurally connected in depth. Grammars and analyzers for such languages are complex and the problems encountered cannot always be solved by the techniques presented here.

CHAPTER IV

DEFINING THE SYNTAX OF A LANGUAGE

4.1 Introduction

The previous chapter explained that a grammar is defined by a set of rules or productions. One of the first steps in the construction of an automatic parsing algorithm is the specification of the syntax. The syntactic specifications for a language provide a set of definitions for the various syntactic units which can be used in the language. A definition is a string of characters and syntactic units.

A syntactic specification of a language is a concise and compact representation of the structure of that language, but it is merely that - and does not constitute a set of rules either for producing allowable strings in the language or for recognizing whether or not a proffered string is in fact an allowable string. The parsing of a proferred string is performed by syntactic analyzer algorithms which use the syntactic specifications and the string as input and produce some representation of the structure of the statement, if possible. (Cheatham (1964))

4.2 Metalanguages

Metalanguages are used to provide an orderly and compact listing of the rules which specify the syntax of the language. Since syntax specifications are essential to parsing algorithms,

the design of an algorithm can be influenced by the metalanguage utilized. Gorn (1961b) describes a number of techniques for specifying a grammar. Some of the methods are suitable for the generation of statements in the language while others simplify the process of recognizing legitimate statements of the language. The systems described by Gorn range over natual languages and subsets thereof, logical expressions, networks, matrices, tree notation, and flow diagrams. A particular method is chosen on the basis of its ability to clarify the concepts involved and simplify the algorithms which will use the specifications. The ease of modifying the specifications is also important.

The following sections contain descriptions of selected metalanguages. Figures 1 through 6 are examples of these metalanguages.

4.3 Backus Normal Form

The most common metalanguage is Backus Normal Form (BNF). It was developed by J.W. Backus and first appeared in the ALGOL 60 Report. BNF is a subset of natural language. It is easily interpreted by software designers and can be coded for use by parsing algorithms.

In the original BNF system, metalinguistic formulae are constructed from the following conventions: Metalinguistic variables, whose values are sequences of symbols, are repre-

sented by sequences of characters enclosed in brackets $\langle \; \rangle$. The marks ::= (equivalent to) and $|$ (or) are metalinguistic connectives. In a formula, any mark which is not a metalinguistic variable or connective, denotes itself (or the class of marks which are equivalent to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Usually the symbols within brackets $\langle \; \rangle$ are chosen to be words describing approximately the nature of the corresponding variable. The original version of BNF has been modified to meet various requirements.

Irons (1963b) implemented changes in BNF to facilitate its use in programming systems. The first of these was the removal of left recursion so that no definitions of the forms

$$
\begin{array}{llll}
\langle A \rangle & ::= & \langle A \rangle \langle B \rangle & \quad \text{or} \\
\langle A \rangle & ::= & \langle B \rangle \langle C \rangle & \\
\langle B \rangle & ::= & \langle A \rangle \langle D \rangle &
\end{array}
$$

were allowed. To offset this restriction, an "iterative power" was introduced. Any set of metalinguistic variables enclosed by the braces { } is specified to occur zero or more times in an input string. The restriction that the brace { may not occur immediately after ::= , must obviously be applied.

<LETTER> ::= A|B|C|----|Z

<DIGIT> ::= 0|1|----|9

<MULOP> ::= ×|÷

<ADDOP> ::= +|-

<VARIABLE> ::= <LETTER>|<VARIABLE> <LETTER>

<INTEGER> ::= <DIGIT>|<INTEGER><DIGIT>

<FACTOR> ::= <VARIABLE>|<INTEGER>| (<ARITH EXPR>)

<TERM> ::= <FACTOR>|<TERM><MULOP> <FACTOR>

<ARITH EXPR> ::= <TERM>|<ARITH EXPR><ADDOP><TERM>

<ASSIGNMENT> ::= <VARIABLE> = <ARITH EXPR>

<PROGRAM> ::= <ASSIGNMENT>|<PROGRAM>;<ASSIGNMENT>

Figure 1.

ORIGINAL BACKUS NORMAL FORM

```
<LETTER>       ::=  A|B|----|Z
<DIGIT>        ::=  0|----|9
<MULOP>        ::=  ×|÷
<ADDOP>        ::=  +|-
<VARIABLE>     ::=  <LETTER> {<LETTER>}
<INTEGER>      ::=  <DIGIT> {<DIGIT>}
<FACTOR>       ::=  <VARIABLE> | <INTEGER> | (<ARITH EXPR>)
<TERM>         ::=  <FACTOR> {<MULOP><FACTOR>}
<ARITH EXPR>   ::=  <TERM> {<ADDOP><TERM>}
<ASSIGNMENT>   ::=  <VARIABLE> = <ARITH EXPR>
<PROGRAM>      ::=  <ASSIGNMENT> {;<ASSIGNMENT>}
```

Figure 2.

IRONS' NOTATION

A subset of ALGOL was described in both Backus notation and Irons' notation. The two descriptions of the grammar are given in the Appendix B.

Iverson (1964) added conventions to BNF to provide a mode of description which is more compact and easier to prepare and use than standard BNF descriptions. The new conventions are:

1. to number the syntactic definitions sequentially and use the sequence number, rather than the name, in all references. Single letter mnemonics are used for the basic alphabet, i.e. terminal symbols.
2. to use an asterisk to denote the syntactic form being defined. Thus a recursive definition, in which a syntactic unit is defined in terms of itself, involves an asterisk.
3. to denote any set of symbols by enclosing the list of symbols in braces { } and apply the set operators $\cup$ (union) and Δ (difference) to any set or syntactic class.
4. to list all synonyms separately and supply only one definition.

4.4 Tree Notation and Related Forms

Tree notation can be used to represent either the grammar of a language or the structure of a statement. Trees consist

Reference	Name	Definition
d	digit	0|1|----|9
ℓ	letter	A|B|C----|Z
1	variable	ℓ| * ℓ
2	integer	d| * d
3	factor	1|2|(5)
4	term	3| * {×÷} 3
5	arith expr	4| * {+-} 4
6	assignment	1 = 5
7	program	6| * ; 6

FIGURE 3

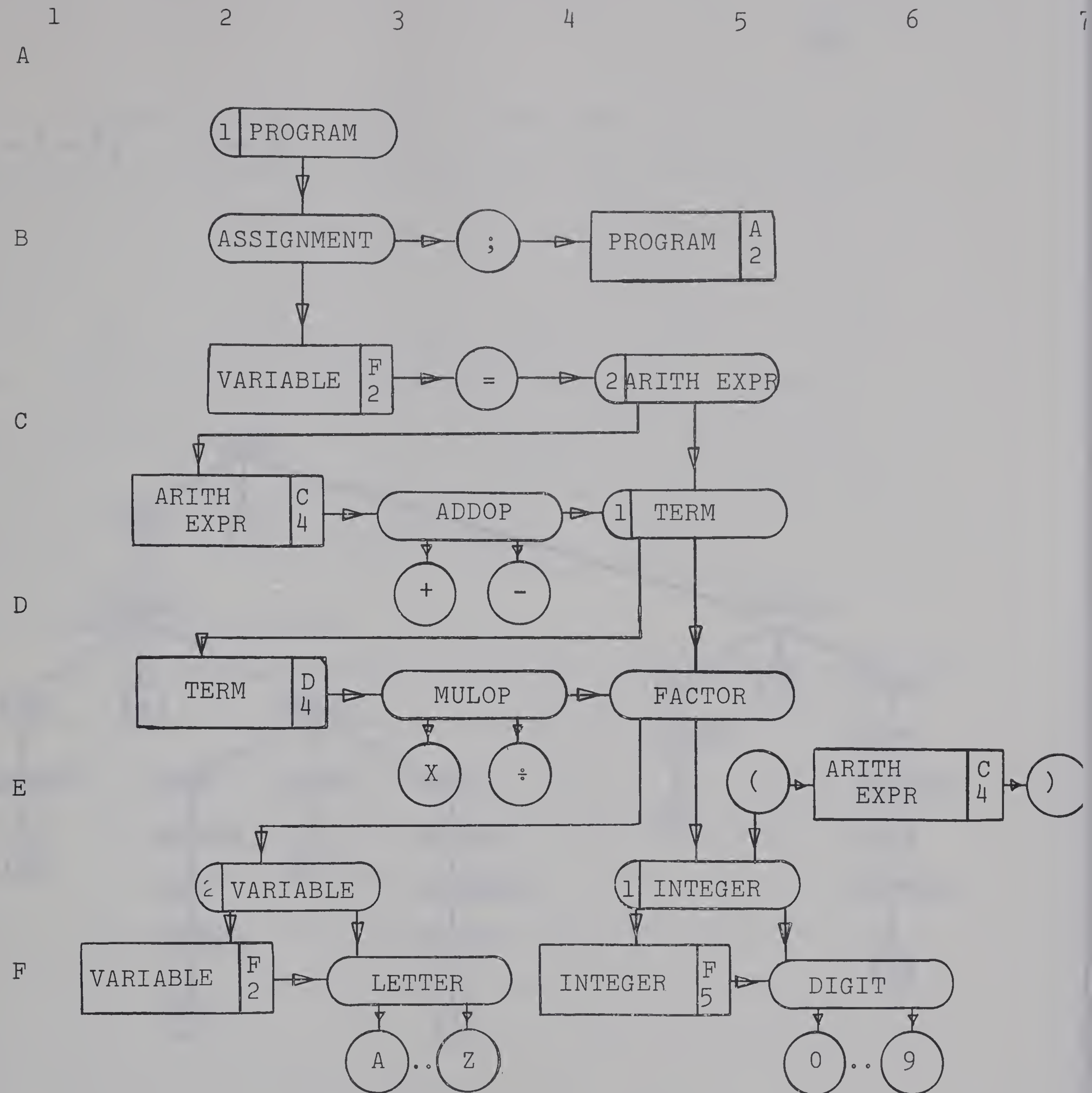
IVERSON'S SPECIFICATIONS

of terminal and non-terminal nodes which are connected by directed paths called branches.

Trees can be classified according to the direction of the connecting branches. A "top down" tree consists of a main node or root from which branches descend to other nodes. The other nodes have entrance branches but do not require exit nodes. Nodes with no exit branches are referred to as terminal nodes while nodes having both entrance and exit branches are non-terminal nodes. The information on a tree may be contained in the nodes, branches, or both. A "bottom up" tree is similar except the branches are directed from the terminal nodes to the root.

Taylor, Turner and Waychoff (1961) converted the conventional BNF notation into tree notation or syntactical charts in order to condense the specifications and simplify the cross referencing of definitions. In this system the shapes of the enclosures and the directions of the connective arrows have special meanings. Charts have not yet been used in automatic parsing algorithms. This is probably due to difficulties in representing them for the algorithms.

When trees are used to represent a parse of a statement, there is only one possible way of connecting the variables, i.e. there can be no alternatives. In this case the nodes themselves can be used to represent the symbols involved.



enclose all terminal characters

indicate all the relevant definitions

connect the basic symbols and metalinguistic variables which form a definition

a metalinguistic variable is defined at this point

the number of other occurrences of this metalinguistic variable is given to the left of |

the definition of the enclosed metalinguistic variable is given at the point of the coordinates to the right of |

FIGURE 4

SYNTACTICAL CHART

A = B + C;   C = D

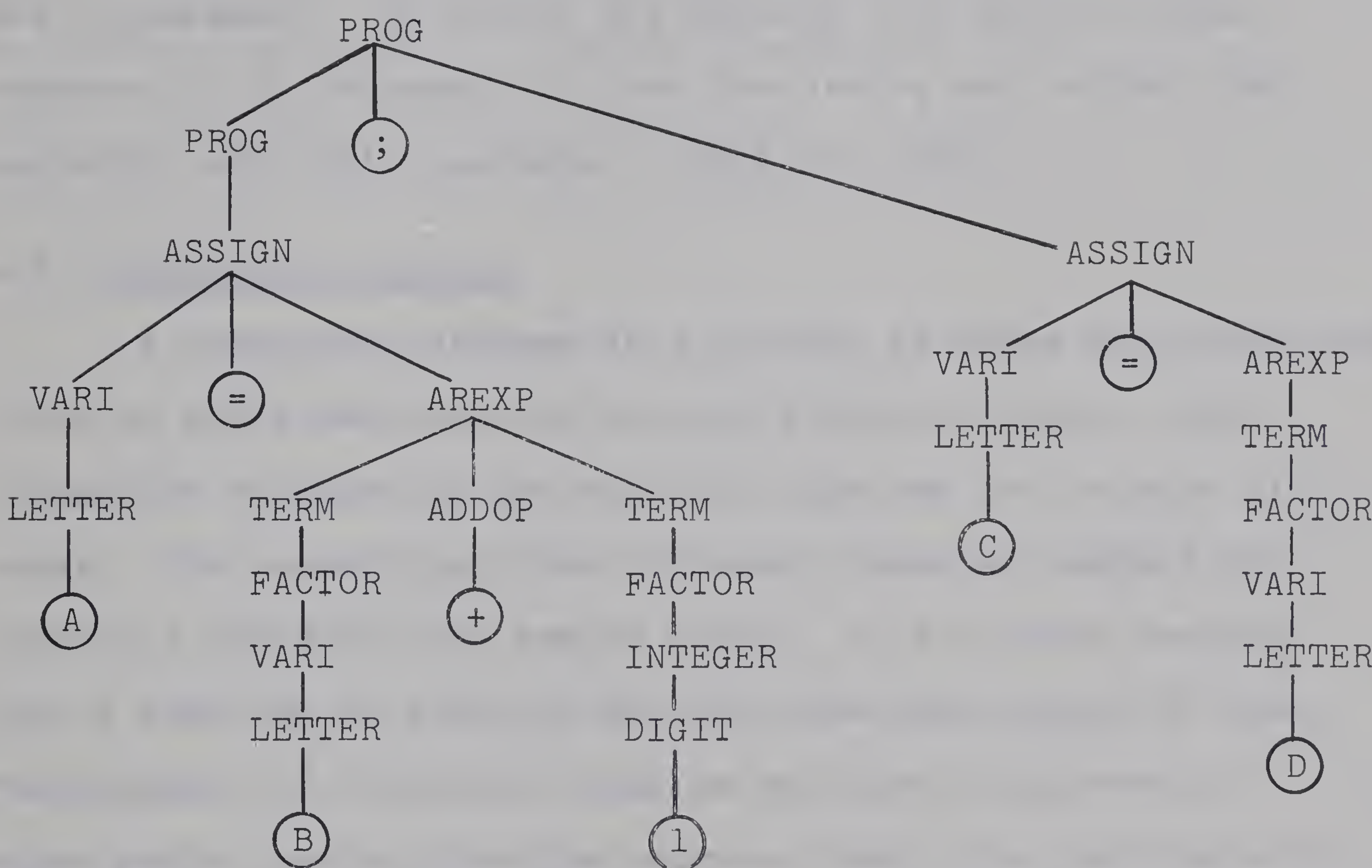


FIGURE 5

PARSE OF A PROGRAM

Graham (1964) presents a number of methods for representing the information contained in a tree through use of matrices and linear sequences. The linear sequences are related to Polish Notation. In this notation the structure of the tree is represented by the symbols involved, coupled with special operators representing alternation, catenation, and replacement. To derive the meaning from such a linear sequence it is necessary to scan the string and collect the operators and their operands. (Hamblin (1962))

4.5 Transition Diagrams

A transition diagram is a network of nodes and connecting lines or paths defining one or more syntactic units. Each transition diagram has one entrance node and one or more exit nodes. The connecting lines represent terminal symbols or syntactic units or they may be blank. No two paths leading from a node may be blank or may have the same symbol on them. Furthermore, no transition diagram may have a sequence of blank paths leading from the entrance node to an exit node nor may a set of blank paths contain a loop.

Two restrictions are placed on transition diagrams to make them useful in translators. The "No Loop Condition" states that no transition diagram will make reference to itself without first processing a terminal character. The "No Backup Condition" requires that once a symbol is read, the syntactic unit of which it is part can be determined without looking back in the input string.

Transition diagrams are easily understood by systems designers and readily coded for use by a translator. However, the design of diagrams is neither straightforward nor easy to describe. (Conway (1963))

4.6 Syntax Specification Difficulties

Languages which can be readily parsed are context independent or of Type 2 and the grammars for these languages can be specified in Backus Normal Form. However, in the construction of syntax oriented compilers specification problems occur because:

1. programming languages are not strictly Type 2,
2. semantic clarity and unambiguity are necessary requirements, and
3. the analysis algorithms must have the specifications in the proper form.

Although the structure of a programming language can be specified in BNF, non-syntactic rules cannot. One such rule is that a variable cannot denote two or more distinct data structures in a program. One solution to this problem involves using two sets of specifications - one supplying the syntax and the other the non-syntactic rules.

Some statements may introduce a context dependent aspect into the analyses. For example, declaration statements inform

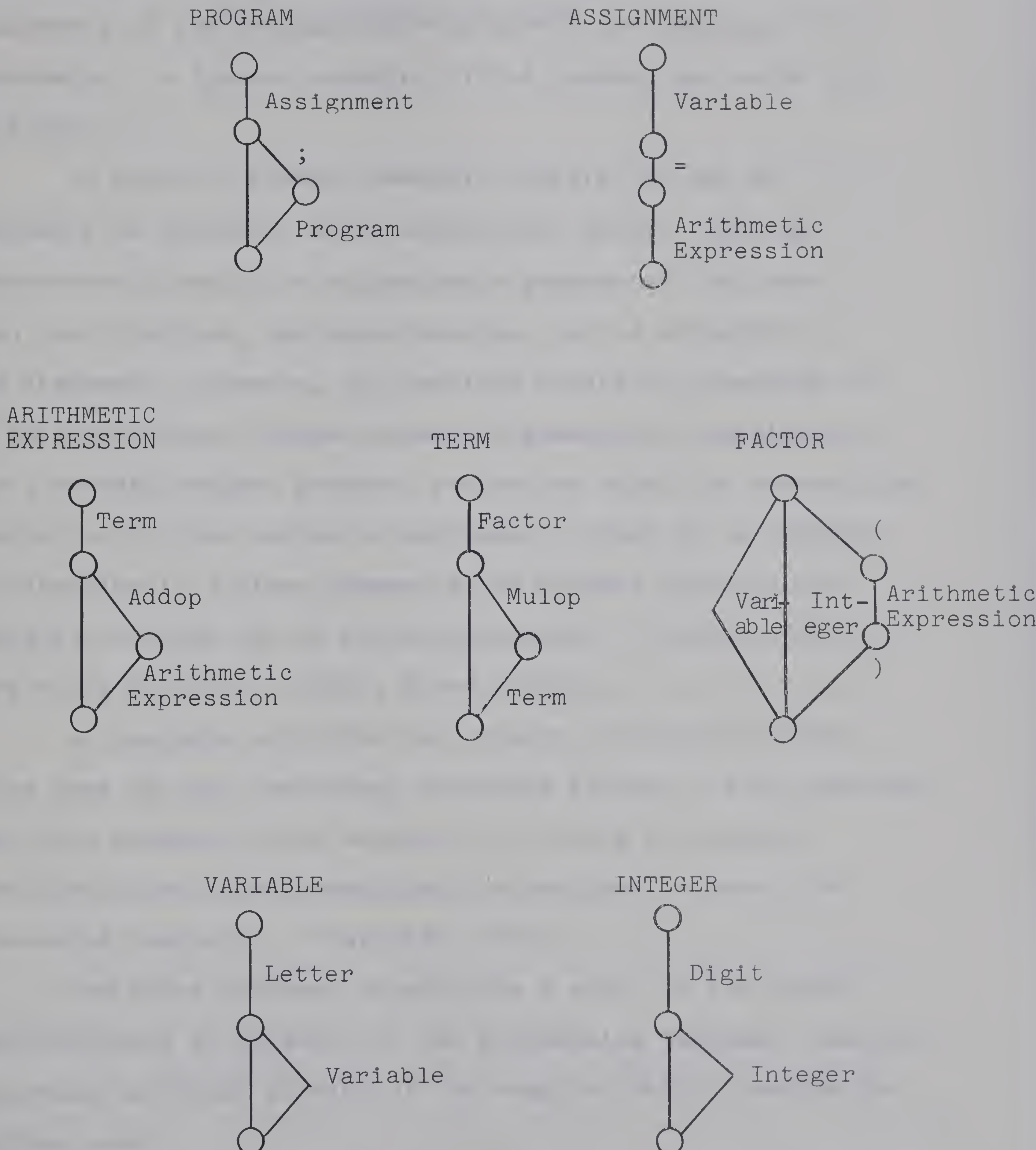


FIGURE 6

TRANSITION DIAGRAMS

the analyzer as to what variables can appear in subsequent statements of the program and thus affect the meaning of the statements. A dynamic changing of the grammar may solve this problem.

In order to produce semantic clarity, it may be necessary to introduce extra syntactical units. This may inadvertently result in an ambiguous grammar so that more than one structure, and hence meaning, may be assigned to one statement. However, no algorithm exists to determine if an arbitrary Type 2 phrase structure grammar is unambiguous. For a bounded context grammar, procedures exist for determining whether or not the system is ambiguous. Since it is possible to determine if a given grammar is of bounded context, the ambiguity problem can be solved indirectly. (Davis (1966), Caracciola di Forino (1963), Floyd (1962))

An analysis algorithm may require the specifications to be free of left recursion, described earlier. Also, problems may arise because of the necessity of having the syntax specified in an order permitting the analyzer to parse the statements correctly. (Metcalfe (1964))

The above problems necessitate a study of the syntax specifications in relation to the programming language, analysis algorithm, and that portion of the compiler which produces the machine code.

CHAPTER V

SYNTAX ORIENTED COMPILERS

5.1 Compilers

In a discussion of the general properties of programming language processors, Davis points out that any processor must

1. linearly scan the source program to recognize and code the symbols used,
2. determine the syntactic structures by isolating all syntactic types based on the syntactic specifications provided,
3. discover all syntactic ambiguities and violations of vocabulary or grammatical rules and act accordingly,
4. translate the program from source to target language based on the structure and semantic specifications provided,
5. optimize the target language for the particular machine, and
6. produce the machine code program.

The three classes of compilers are conventional, syntax oriented, and list processing compilers.

The conventional approach has the language and machine specifications programmed into the compiler algorithm. Although it appears this approach produces the fastest compiler and most

efficient machine code, changes in the source language necessitate changes in the compiler program.

List processors utilize rules governing the form, content, and linkages between variable length pieces of data called lists. No rigid syntactic specifications are used. A program in a list processing language is a series of actions to be performed on lists. Because it is not possible to permanently allocate data storage, list processors only determine the structure of the program and set up tables of processing routines to handle the lists.

Syntax oriented compilers utilize tables to supply the required information about the source and target languages in the compiling process. These compilers are easy to write and simplify the requirements for making changes in the languages. However, it is reasonable to assume that the speed of compilation and efficiency of the machine code produced are inversely related to the number of restrictions placed on the grammar.

5.2 A General Syntax Oriented Compiler

The simplicity and generality of the syntax oriented approach are two strong reasons for using it in the construction of compilers for all programming languages. A syntax oriented compiler for translating programs in a source language $\underline{L}$ to machine code for a machine $\underline{M}$ requires the following preliminary steps before becoming operational.

1. The formal syntax of L is loaded into the system and the representation of the grammar required by syntactic analyzer is developed.
2. The semantics or meaning of the syntactic structures of L in terms of the machine M are loaded and processed.

When in operation, the compiler will translate syntactic constructs to semantic constructs based on the formalization of syntax and semantics, but not on any particular representations. The basic outline of such a system is given in Figure 7. Only the part to the right of the double line is required for the compiler. (Feldman (1966))

A syntax oriented compiler has five major components:

1. The loader converts the source program to a representation used by the compiler. It can also perform minor functions such as removal of comments, and detection of simple errors.
2. The syntactic analyzer parses the program, detects and processes errors, and produces a syntactic representation of the program in tree notation.
3. The generator is a set of tables containing a generator strategy to be applied at each node of the tree. The generator strategies describe each type of node and list the actions to be taken. The

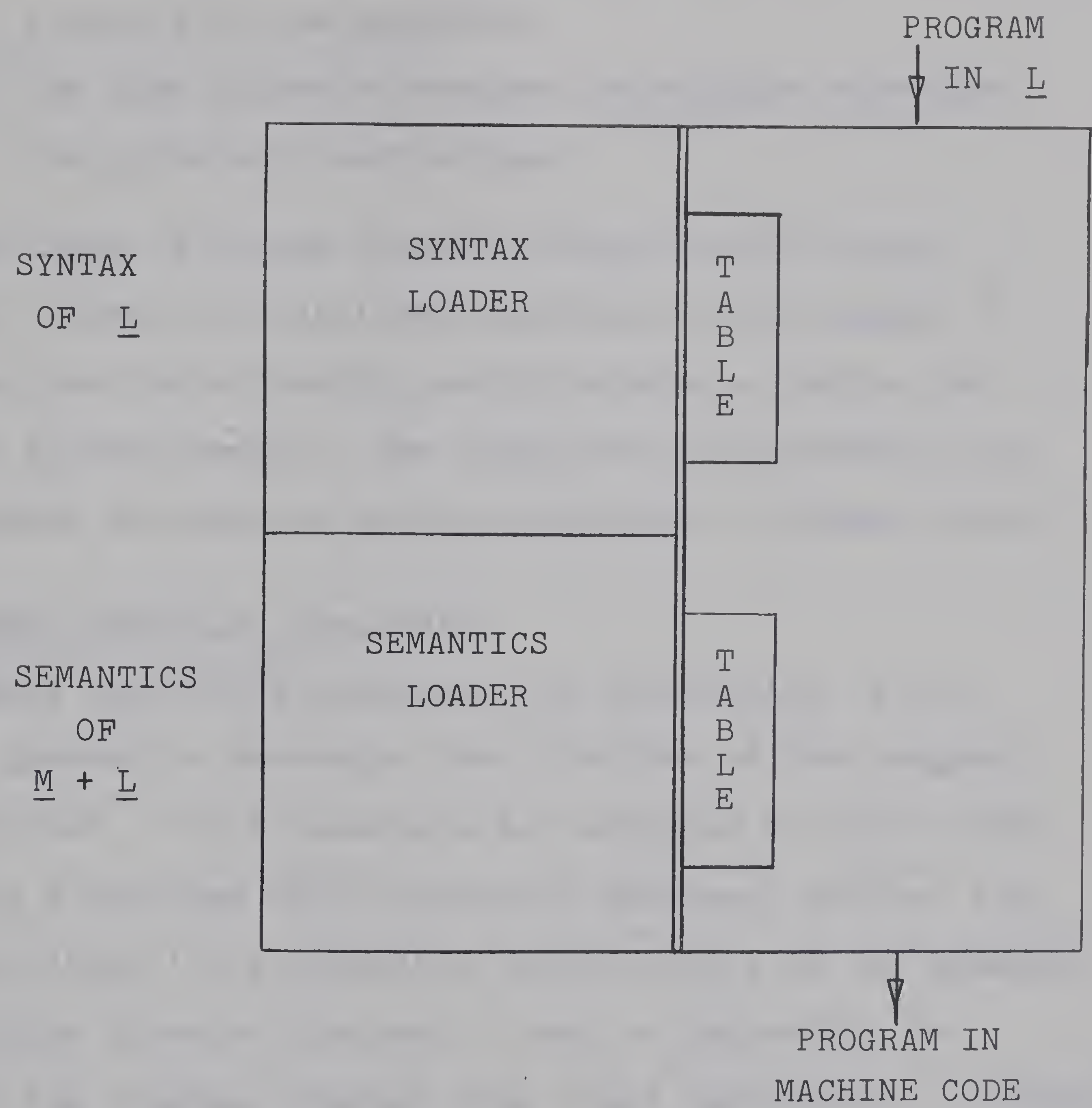


FIGURE 7

A BASIC SYNTAX ORIENTED COMPILER

actions are either to proceed to the neighbouring node or produce the macro instructions.

4. The macro accumulator optimizes the instructions produced by the generator.
5. The code selector produces the machine code from the optimized instructions.

Two types of syntax oriented compilers have been developed. They are classified according to the manner in which they use the syntactic specifications to derive the structure of the program. The compilers are referred to as either syntax directed or syntax controlled. (Graham (1964))

5.3 Syntax Controlled Analyzers

Syntax controlled analyzers use tabulations of the original grammar to determine the structure of the program being compiled. The tabulations are produced by preliminary processing algorithms which construct matrices, tables, and lists describing the permissible constructions of the grammar. Unlike syntax directed systems, it may be impossible to construct the original grammar from these tabulations. Parsing algorithms for this class can be considered as non-predictive in that known parameters are used to decide what actions are to be taken. Syntax controlled analyzers have been developed by Floyd (1963) and Wirth and Weber (1966); Evans (1964) and Feldman (1966); and Eickel, Paul, Bauer and Samelson (1963).

As these analyzers are applicable to restricted grammars such as operator and precedence grammars, they will not be considered further (Galler et al (1967)).

5.4 Syntax Directed Analyzers

Syntax directed analysis is any procedure which is capable of constructing a syntax tree for an arbitrary program in an arbitrary phrase structure language. The syntax tree is a structured representation of the information contained in the source program and indicates the relationships among the syntactic units formed by the terminal characters. In a compiler, suitable processes translate the tree into a machine code program or derivation tree for an equivalent program in another language.

In the production of the syntax tree, syntax directed analyzers use a complicated hierarchy of goals in an attempt to attain a principal goal, i.e. a program. Two general methods are possible. Top-down parsing begins by looking for the principal goal and then substitutes subordinate goals. Left recursive definitions may pose problems for this approach as the possibility of an infinite number of subordinate goals arises. Bottom-up parsing begins by considering the terminal characters and attempts to construct the higher elements. In each of these processes, predictions are made as to how the program is constructed. If a prediction proves false at some stage, a new prediction is made. A history of the successful

predictions supplies the required syntax tree. (Floyd (1964b), Cheatham and Sattley (1964))

Normally, longer programs require longer analysis times. For syntax directed systems it is not known in general whether this increase is exponential or not. However, techniques similar to those used by human analysts can be incorporated into the algorithms to reduce the number of attempts resulting in incorrect structures. Conway (1963) limits the choices of alternatives by examining the first character or word of the constructions. Ingerman (1966) and Irons (1963b) use matrices which indicate if a particular terminal character can occur in a specified definition. (Floyd (1964b))

Models of syntax directed compilers and examples of parsed statements are given in the appendix.

5.5 Advantages

The ease of compiler construction and ease of introducing changes in the source language are the main reasons for attempting to develop syntax directed compilers. However, additional advantages have become apparent with the development of such compilers.

One benefit arises from the necessity of adequately specifying the syntax. This requirement should improve programming languages by removing most ambiguities before the language is released for general use and by simplifying corrections when they are required. Furthermore, analyzers

capable of determining all possible parses of a string can detect ambiguous definitions in a programming language, as two or more parses would result. (Irons (1963b)) A related advantage is that a complete syntax directed compiler for a particular source language computer combination would provide a rigorous and complete documentation of the languages (source and target) along with the method of translation involved. (Metcalfe (1964))

Graham (1964) contends that most optimization algorithms such as elimination of common subexpressions and optimum evaluation of Boolean expressions are much simpler when applied to some intermediate form rather than to the original expression or the final machine language version. Iverson (1962) describes a process for reducing Polish notation to a form in which the number of operands and operators is a minimum.

Leavenworth (1966) describes a translation approach which allows one to extend the syntax and semantics of a given high-level base language by the use of syntax macros. The syntax macros are used to define new statements and expressions by means of the syntactic units in the base language rather than machine instructions. A syntax macro has two parts:

1. A macro structure which describes the syntax of the source text to be recognized;
2. A macro definition which describes the semantics of the corresponding macro structure in terms of the

base language. When the compiler recognizes a unit defined by a syntax macro, it expands the unit into the source code presented in the macro definition for further translation. Thus the flexibility of the base language can easily be increased without new semantic definitions in machine language and without the addition of special symbols.

5.6 Disadvantages

The implementation of practical syntax oriented compilers has been delayed by a number of problems. Processing syntactically incorrect strings is foremost among these. The compilers can usually detect the existence of errors but may have difficulty in pinpointing the exact location and type of error. As a result, meaningful diagnostic routines are difficult to construct. Furthermore, after an error is encountered it may be difficult to correct the effect of the error on the compiler sufficiently to enable the system to analyze subsequent parts of the input string. (Irons (1963a))

A number of algorithms have been proposed to solve this problem. Conway's transition diagrams utilize a "NO BACKUP" condition on the input string. The programmer then knows how far the compiler was able to proceed before failing. Special "error" syntactic units can also be defined. These units would be established as correct goals in case of program errors.

Irons (1963b) developed a multi-parse syntactic analyzer which preserves all possible parses for a program. An error condition reduces the number of existing parses to zero at the error or shortly after.

Problems arising from context dependent features of a language were mentioned in relation to the methods of specifying the syntax. Declaration statements are an example of context dependent statements. Normally these do not require the generation of executable code nor do they modify the structure of the program. However, they may change the syntax through coded information in the symbol table for the program variables. (Cheatham et al (1964))

Although the intermediate form produced by syntax oriented compilers aids the optimization of coding, it is still necessary to do a large amount of work to produce highly efficient code. Automatic parsing algorithms supply the structure of a program but attempts must be made to simplify the description of the structure as an aid in the production of "optimized" machine code (Irons (1963a), Cheatham et al (1964)).

5.7 Non-Compiler Applications

Although syntax oriented techniques have been developed to parse statements in programming languages, a number of different applications have been suggested and experimented with.

Metcalfe (1964) suggested that requests to information storage and retrieval systems be expressed in a restricted form of a natural language. A syntax analyzer could convert the natural language to instructions which the system would use to search for the requested information.

Problems of data input for digital computers may also be solved by syntax oriented techniques. The interpretation of free and fixed format control cards is a possible example. Raphael (1966) proposed the use of modified syntax analyzers as a means of translating input from graphic display devices and generating the necessary machine instructions.

Automatic algebraic manipulations may also be implemented through these processes. Schon (1965) developed an algorithm which could perform analytic differentiation using syntax oriented techniques.

Parsing algorithms can be applied to any structure, whether it be linear, spatial, temporal, or otherwise. Kirsch (1964) points out that the two main methods for recording information - pictures and text - are closely related. Thus, computers may interpret pictures through syntactic descriptions and provide a coded text-oriented description. This idea is similar to one proposed by Narasimham (1966) in which statements are used to describe the various aspects of a picture or object.

The descriptions are built up through the use of attributes and primitive objects. Primitive objects, the constituent parts of the pictures, correspond to the terminal characters of languages. Attributes are the characteristics which pictures could have and define the non-terminal units. Irons (1963a) proposed combining a multiparse algorithm with a pattern recognition device to offer several interpretations of a picture and a weight for each. These techniques could be applied to the analysis of bubble chamber pictures, letter recognition problems, etc.

CHAPTER VI

THREE SYNTAX DIRECTED ANALYZERS

The remainder of this thesis is concerned with a description of three algorithms for parsing strings in relation to a set of syntactic specifications. These algorithms can be used as the second major component of syntax directed compilers or as the basis of systems suggested in Section 5.7, Non-Compiler Applications.

Although Backus Normal Form (or a modification thereof) is easily understood by compiler designers, it is of little direct value to an automatic analyzer. Thus, an essential step in the construction of a parsing algorithm is preprocessing the BNF.

Since computer algorithms operate most efficiently with numbers, the prose descriptions of BNF are converted to numerical codes. A number of utility routines based on a paper by Williams (1959) facilitate this conversion process. These numerical codes are processed by algorithms which produce tables to be used by the syntax directed analyzers.

The analyzers given here include a conventional analyzer, a multiple parse analyzer and a transition diagram analyzer. The descriptions and listings of the programs along with sample runs are given in the appendices.

6.1 A Conventional Analysis Algorithm

One of the first syntax directed compilers was developed by Irons (1961). His parsing algorithm required the syntax to be specified in semi-linked lists. To increase processing speed, he developed an acceptance matrix which indicated if a particular metacomponent could be developed as a first element in the definition of a metaresult. Cheatham and Sattley (1964) developed a related system which is capable of processing left recursive definitions. However, this system does not use an acceptance matrix. Ingermann, in his book, "A Syntax-Oriented Translator" describes a similar analyzer and provides an algorithm for developing an acceptance matrix. The algorithm presented here is Cheatham's system modified to take advantage of an acceptance matrix.

6.2 A Transition Diagram Algorithm

In an effort to overcome problems of compilation speed and error detection, Conway (1963) developed a translation system using transition diagrams. The structure of a program is obtained by starting at the entrance node of a main diagram (i.e. PROGRAM) and recording which lines and diagrams are traversed while attempting to reach its exit node by matching characters in the input string with terminal characters in the diagrams. This algorithm is of particular interest because it is being used in the APL interpreter.

6.3 A Multiple Parse Algorithm

Irons (1963b) also developed a multiple parse syntactic analyzer to improve error detection in syntax directed compilers. The preprocessing phase determines the terminal characters, the syntactic units these characters can initialize, and the units which must be present to complete the initialized units. The analyzer operates by considering each input symbol and determining all feasible parses of the string up to that point.

The three systems for which models are constructed have been chosen because of their generality, type of syntax specifications used, and explicit representation of the parses provided. All methods use the most general form of phrase structure grammars. This reduces the number of restrictions which must be observed in specifying the syntax of a language and still permits the algorithms to be applied to more restricted grammars. The preprocessing algorithms convert syntax specifications in BNF or Irons' notation to vectors and matrices which are closely related to the original form. Consequently it is easier to understand the operation of the algorithms and develop the models.

The conventional and transition diagram algorithms produce the first correct parse which results from the particular order in which the grammar is specified. The multiple parse algorithm produces all correct parses. The output of each system can be converted to tree notation, thus simplifying interpretation of the parses.

CONCLUSION

All of the models determined the structure of simple arithmetic statements. The conventional and multiple parse algorithms also analyzed statements formed in a subset of ALGOL. The development and testing of the models produced general results concerning their construction, syntax specification requirements, error detection facilities, efficiency, and usage. The work also served as one evaluation of APL for model building.

The preprocessing routines for the conventional and multiple parse algorithms are straightforward. It was impossible to write algorithms which would construct transition diagrams for the subset of ALGOL. The matrices produced by the multiple parse preprocessing routines appear to contain the necessary information and the problem entails designing diagrams which satisfy the "No Loop" and "No Backup" conditions. It is possible to construct the diagrams manually although this was not done for the ALGOL example. The analysis routines require careful consideration in their construction as minor changes can have considerable effect on the routines themselves as well as the preprocessing routines.

The algorithms are of most value if the syntax is specified in such a manner that the system designer can understand it and the algorithms can process it. Although both BNF

and Irons' notation are readily understood by designers, Irons' notation is more useful for syntax directed systems. In the examples considered it is at least as powerful as BNF and is a more concise representation of the syntax. Although the conventional routines used BNF they would have been simplified by the use of Irons' notation; BNF would not simplify the other algorithms.

A syntax directed analyzer must determine the structure of statements efficiently in terms of speed and storage requirements. No tests were conducted to determine the quantitative aspects of speed and storage requirements, but qualitative judgements can be made. The transition diagram algorithm would be the fastest as it always determines part of the structure with each recognition of a terminal character. The conventional algorithm is slower as it may have to reconsider input characters if a false structure is attempted. The multiple parse algorithm is slowest as it must consider all parses. The transition diagram algorithm is most efficient because the structural connections in the language are determined in the preprocessing phase and not in the analyzer algorithm.

The transition diagram algorithm requires the least storage followed by the conventional and multiple parse systems. The transition diagram algorithm requires the syntax specifications and one working vector. Information concerning the parses

need not be stored in the machine as there is no possibility of false structures. The conventional routines require the syntax specifications and three working vectors. Information related to the parses may have to be stored because of false structures. The multiple parse algorithm requires extensive tables describing the syntax specifications and storage for parses $n - 1$ and n if character n is being considered. In both the conventional and multiple parse algorithms, auxiliary storage could be used to retain the parse information. However, this would decrease the speed of the algorithms.

All programs which are subjected to analysis are not correct and the algorithms must be able to locate the errors. In this respect the algorithms are essentially equivalent. The transition diagram and multiple parse algorithms indicate which input symbol they were considering when they were unable to produce further output. The conventional algorithm indicates the character farthest along the input string which it had processed before coming to an error. No attempts were made to have the algorithms process the input string beyond the error condition.

The algorithms could be used in any situation requiring the determination of structure providing a suitable means of coding the input could be devised. The particular algorithm used would depend on the requirements of the problem. The transition diagram system offers the best syntax directed analyzer. Until algorithms which will produce the diagrams

from the syntax specifications are available, much of the flexibility of such systems will be mssing. The development of these preprocessing routines would constitute a significant achievement in this field.

The construction of models using APL in a timesharing environment produced better returns than could be expected from conventional, batch processed programming languages. The powerful instruction set reduced the amount of coding required and thus reduced the number of mechanical errors. The time-sharing environment permitted rapid correction of the errors which did occur. APL would be more useful for such work if it had an iterative instruction, facilities to insert comments, and a better arranged listing of the programs. These features would make it easier to study and understand the models.

Syntax directed analysis started as a means to analyze the structure of statements in a programming language. Before syntax directed compilers will compete with conventional compilers many problems must be overcome. Consequently, the first large scale use of syntax directed analysis may be in the solution of non-compiler problems. If there is a demand for programming languages with complex grammars, syntax directed analyzers can provide the basis for simple, flexible compilers.

BIBLIOGRAPHY

Caracciola di Forino, A., 1963, "Some remarks on the syntax of symbolic programming languages", Comm. ACM, 6/8, pp. 456-460.

Cheatham, T.E. Jr. and Sattley, K., 1964, "Syntax-directed compiling", AFIPS Conf. Proc., vol. 25, pp. 31-57.

Chomsky, N., 1959, "On certain formal properties of grammars", Information and Control, vol. 2, pp. 137-167.

Conway, M.E., 1963, "Design of a separable transition-diagram compiler", Comm. ACM, 6/7, pp. 396-408.

Davis, R.M., 1966, "Programming language processors", Advances in Computers, ed. Alt, F.L., Academic Press, New York: vol. 7, pp. 117-180.

Eickel, J., Paul, M., Bauer, F.L., and Samelson, K., 1963, "A syntax controlled generator of formal language processors", Comm. ACM, 6/8, pp. 451-455.

Evans, A., 1964, "An ALGOL-60 compiler", Ann. Rev. in Auto. Prog., vol. 4, pp. 87-123.

Feldman, J.A., 1966, "A formal semantics for computer languages and its applications in a compiler-compiler", Comm. ACM, 9/1, pp. 3-12.

Floyd, R.W., 1962, "On ambiguity in phrase structure languages", Comm. ACM, 5/10, pp. 526-534.

Floyd, R.W., 1963, "Syntactic analysis and operator precedence", JACM, 10/3, pp. 316-333.

Floyd, R.W., 1964a, "Bounded context syntactic analysis", Comm. ACM, 7/2, pp. 62-65.

Floyd, R.W., 1964b, "The syntax of programming languages - a survey", IEEE Transactions on Electronic Computers, vol. EC-13, pp. 346-353.

Galler, B.A. and Perlis, A.J., 1967, "A proposal for definitions in ALGOL", Comm. ACM, 10/4, pp. 204-219.

Gorn, S., 1961a, "Some basic terminology connected with mechanical languages and their processors", Comm. ACM, 4/8, pp. 336-339.

Gorn, S., 1961b, "Specification languages for mechanical languages and their processors. -A baker's dozen", Comm. ACM, 4/9, pp. 532-542.

Graham, R.M., 1964, "Bounded context translation", AFIPS Conf. Proc., vol. 25, pp. 17-29.

Greibach, S., 1965, "A new normal-form theorem for context-free phrase structure grammars", <u>JACM</u>, vol. 12, pp. 42-52.

Hamblin, C.L., 1962, "Translation to and from polish notation", <u>Comput. J.</u>, vol. 5, pp. 210-213.

Ingerman, P.Z., 1966, <u>A Syntax-Oriented Translator</u>, Academic Press, New York: 131 pp.

Irons, E.T., 1961, "A syntax directed compiler for ALGOL 60", <u>Comm. ACM</u>, 4/1, pp. 51-55.

Irons, E.T., 1963a, "The structure and use of the syntax directed compiler", <u>Ann. Rev. in Auto. Prog.</u>, ed. Goodman, R., MacMillan Comp., New York: vol. 3, pp. 207-228.

Irons, E.T., 1963b, "An error correcting parse algorithm", <u>Comm. ACM</u>, 6/11, pp. 669-674.

Irons, E.T., 1964, "Structural connections in formal languages", <u>Comm. ACM</u>, 7/2, pp. 67-72.

Iverson, K.E., 1962, <u>A Programming Language</u>, John Wiley and Sons, Inc., New York: 286 pp.

Iverson, K.E., 1964, "A method of syntax specification", <u>Comm. ACM</u>, 7/10, pp. 588-589.

Kirsch, R.A., 1964, "Computer interpretation of English text and picture patterns", _Trans IEEE_, vol. EC-13, pp. 363-376.

Kuno, S., 1966, "The augmented predictive analyzer for context-free languages - Its relative efficiency", _Comm. ACM_, 9/11, pp. 810-823.

Kurki-Suonio, R., 1966, "On top-to-bottom recognition and left recursion", _Comm. ACM_, 9/7, pp. 527-528.

Landweber, P.W., 1964, "Decision problems of phrase-structure grammars", _IEEE Trans._, vol. EC-13, pp. 354-362.

Leavenworth, B.M., 1966, "Syntax macros and extended translation", _Comm. ACM_, 9/11, pp. 790-793.

Metcalfe, H.H., 1964, "A parameterized compiler", _Ann. Rev. in Auto. Prog._, ed. Goodman, R., MacMillan Comp., New York: vol. 4, pp. 125-167.

Narasimham, R., 1966, "Syntax-directed interpretation of classes of pictures", _Comm. ACM_, 9/3, pp. 166-173.

Schorr, H., 1965, "Analytic differentiation using a syntax-directed computer", _Comp. J._, vol. 7, pp. 290-298.

Taylor, W., Turner, L., and Waychoff, R., 1961, "A syntactical chart of ALGOL 60", _Comm. ACM_, 4/5, pp. 393-394.

Williams, F.A., 1959, "Handling identifiers as internal symbols in language processors", Comm. ACM, 2/6, pp. 21-24.

Wirth, N., and Weber, H., 1966, "EULER: A generalization of ALGOL, and its formal definitions", Comm. ACM, Part 1 9/1, pp. 13-25, Part 2 9/2, pp. 89-99.

APPENDIX A

DESCRIPTIONS AND LISTING OF ROUTINES

Figure 7 indicates which subroutines are required by other functions.

CONVENTIONAL ROUTINES

ANALYZE PROG

This algorithm determines the structure of an input string *PROG* by using the vectors set up by *CHEATMOD*.

CHEATMOD RULE

This algorithm develops a set of interrelated vectors from the numeric representation of the syntax specifications *RULE*. These must be in BNF.

Syntax Type Table - one entry for each syntactic unit.

INDEX - the numerical representation of the syntactic type.

TERM - = 0 if the unit is a non-terminal character.
= 1 if the unit is a terminal character.

LKFR - if *TERM* = 0 points to row in structure table where definition of this unit began.
- if *TERM* = 1 same value as INDEX.

Syntax Structure Table - one entry for each constituent of a definition.

TYCD - numeric value of element in definition.

STRC - = 0 definition can not be considered complete.

= 1 definition can be considered complete.

SUCC - points to row of structure table which can come next.

ALTR = 0 no alternate to this unit

= $^-1$ alternate possible but not necessary

> 0 row of structure table which may replace this one.

CHEATOUT

This function displays the vectors prepared by *CHEATMOD*.

INGMOD

This function prepares vectors *ROW* and *COL* and an array *MATRIX* which indicates what terminal symbols given in *ROW* can occur in the metaresults given in *COL*. The definitions are provided by the matrix *RULE* which is the numeric representation of the syntax specifications in *BNF*.

INGMODOUT MATRIX

Outputs the vectors *ROW* and *COL* and the array *MATRIX*.

```
      ∇ANALYZE[⎕]∇

    ∇ ANALYZE PROG

[1]   GSTACK←⍳0
[2]   SSTACK←⍳0
[3]   CSTACK←⍳0
[4]   PROGT←IDENTPROGRAM PROG,' ? '
[5]   GOAL←21
[6]   LSTCHAR←1
[7]   SOURCE←0
[8]   CHAR←1
[9]   →ANA2
[10]  ANA1:GOAL←INDEX⍳TYCD[SOURCE]
[11]  ANA2:→(TERM[GOAL]=1)/ANA5
[12]  →(MATRIX[ROW[PROGT[CHAR]];COL[INDEX[GOAL]]]=0)/ANA6
[13]  LOAD 'G'
[14]  LOAD 'S'
[15]  LOAD 'C'
[16]  SOURCE←LKFR[GOAL]
[17]  →ANA1
[18]  ANA5:→(LKFR[GOAL]≠PROGT[CHAR])/ANA6
[19]  ((1+⍴GSTACK);'  ',((T≠' ')/T←,CODE[CHAR;]),' TERMINAL')
[20]  CHAR←CHAR+1
[21]  LSTCHAR←CHAR
[22]  →ANA10
[23]  ANA6:→(SOURCE=0)/ANA15
[24]  →(ALTR[SOURCE]=¯1)/ANA12
[25]  →(ALTR[SOURCE]≠0)/ANA8
[26]  UNLOAD 'G'
[27]  UNLOAD 'S'
[28]  UNLOAD 'C'
[29]  →ANA6
[30]  ANA8:SOURCE←ALTR[SOURCE]
[31]  →ANA1
[32]  ANA10:→(STRC[SOURCE]=0)/ANA13
[33]  ANA11:→(SUCC[SOURCE]=0)/ANA12
[34]  ANA13:SOURCE←SUCC[SOURCE]
[35]  →ANA1
[36]  ANA12:UNLOAD 'G'
[37]  UNLOAD 'S'
[38]  CSTACK←UNSTACK CSTACK
[39]  (1+⍴GSTACK;'  ',(T≠' ')/T←,EXTERNAL[INDEX[GOAL];])
[40]  →(SOURCE≠0)/ANA10
[41]  →0
[42]  ANA15:('ERROR  ';LSTCHAR)
    ∇
```

```
      ∇CHEATMOD[⎕]∇

     ∇ CHEATMOD RULE
[1]    TERM←LKFR←INDEX←⍳0
[2]    TYCD←STRC←SUCC←ALTR←LFRC←⍳0
[3]    C←'BEGIN SWEEP THRU RULES'
[4]    I←0
[5]    NST←1
[6]    C←'CHECK FOR UNPROCESSED LEFT RECURSION'
[7]    CHTM1:→((⍴LFRC)≠0)/CHTM8
[8]    →((⍴RULE)[1]<I←I+1)/CHTM15
[9]    LFT←RULE[I;2]
[10]   FSTCM←NST
[11]   C←'SET UP TYPE TABLE'
[12]   LKFR←LKFR,NST
[13]   TERM←TERM,0
[14]   INDEX←INDEX,LFT
[15]   LSTCM←⍳0
[16]   C←'PREPROCESS THIS RULE'
[17]   LFR←0
[18]   NALT←+/RULE[I;1+⍳RULE[I;1]-1]∊ 1 2
[19]   J←3
[20]   CHTM20:→(RULE[I;1]<J←J+1)/CHTM21
[21]   →(RULE[I;J]=2)/CHTM20
[22]   →(~(2≥RULE[I;J-1])∧LFT=RULE[I;J])/CHTM20
[23]   LFR←J
[24]   →CHTM20
[25]   CHTM21:NRALT←NALT-LFR≠0
[26]   C←'PROCESS RULE I'
[27]   J←3
[28]   NPALT←0
[29]   CHTM2:→(RULE[I;J←J+1]=2)/CHTM2
[30]   C←'CHECK FOR LEFT RECURSION'
[31]   →(J=LFR)/CHTM4
[32]   C←'PREPROCESS THIS COMPONENT'
[33]   COMP←RULE[I;J]
[34]   TYCD←TYCD,COMP
[35]   ALTR←ALTR,0
[36]   NST←NST+1
[37]   C←'CHECK FOR LAST COMPONENT'
[38]   →((NEXT←RULE[I;J+1])≤2)/CHTM3
[39]   C←'NOT LAST COMPONENT'
[40]   STRC←STRC,0
[41]   SUCC←SUCC,NST
[42]   →CHTM2
[43]   C←'LAST COMPONENT'
[44]   CHTM3:LSTCM←LSTCM,NST-1
[45]   NPALT←NPALT+1
```

```
[46]  STRC←STRC,1
[47]  SUCC←SUCC,0
[48]  C←'CHECK FOR LAST ALTERNATIVE'
[49]  →(NEXT=0)/CHTM1
[50]  C←'CHECK FOR LAST NONRECURSIVE ALTERNATE'
[51]  →(NPALT=NRALT)/CHTM2
[52]  FSTCM←ALTR[FSTCM]←NST
[53]  →CHTM2
[54]  C←'DETERMINE EXTENT OF LEFT RECURSIVE DEFN'
[55]  CHTM4:LFRC←LFRC,J
[56]  CHTM7:→((T=0),(T=2),2<T←RULE[I;J←J+1])/CHTM8,CHTM2,CHTM4
[57]  C←'PROCESS LEFT RECURSION'
[58]  CHTM8:K←1
[59]  HAND←NST
[60]  CHTM9:K←K+1
[61]  TYCD←TYCD,TEMP←RULE[I;LFRC[K]]
[62]  ALTR←ALTR,0
[63]  NST←NST+1
[64]  C←'CHECK FOR LAST COMPONENT'
[65]  →(K=ρLFRC)/CHTM11
[66]  SUCC←SUCC,NST
[67]  STRC←STRC,0
[68]  ALTR←ALTR,0
[69]  →CHTM9
[70]  C←'LAST COMPONENT'
[71]  CHTM11:SUCC←SUCC,HAND
[72]  STRC←STRC,1
[73]  SUCC[LSTCM]←HAND
[74]  ALTR[HAND]←¯1
[75]  LFRC←ι0
[76]  →CHTM1
[77]  C←'COMPLETE TYPE TABLE'
[78]  CHTM15:I←4
[79]  CHTM17:→((ρEXTERNAL)[1]<I←I+1)/CHTM16
[80]  →(I∊INDEX)/CHTM17
[81]  INDEX←INDEX,I
[82]  TERM←TERM,1
[83]  LKFR←LKFR,I
[84]  →CHTM17
[85]  CHTM16:'FINI'
```

```
      ∇CHEATOUT[⎕]∇

    ∇ CHEATOUT

[1]   CONTROL←⎕
[2]   I←CONTROL[2]-1
[3]   →(CONTROL[1]=2)/CHTO2
[4]   STOP←⌊/CONTROL[3],ρTERM
[5]   '     SYNTAX TYPE TABLE'
[6]   ' '
[7]   'NUM TYPE  INDEX TERM LKFR'
[8]   CHTO1:→(STOP<I←I+1)/0
[9]   (I;'    ',EXTERNAL[INDEX[I];],(3ρ' ');INDEX[I],TERM[I
      ],LKFR[I])
[10]  →CHTO1
[11]  CHTO2:STOP←⌊/CONTROL[3],ρTYCD
[12]  ' SYNTAX STRUCTURE TABLE'
[13]  ' '
[14]  'INDEX TYCD STRC SUCC ALTR'
[15]  CHTO3:→(STOP<I←I+1)/0
[16]  ('    ';(I,TYCD[I],STRC[I],SUCC[I],ALTR[I]))
[17]  →CHTO3
    ∇


      ∇INGMODOUT[⎕]∇

    ∇ INGMODOUT MATRIX

[1]   'COLUMNS'
[2]   I←0
[3]   INGO2:→((ρMATRIX)[2]<I←I+1)/INGO1
[4]   (I;'    ',EXTERNAL[COLιI;])
[5]   →INGO2
[6]   INGO1:I←0
[7]   ' '
[8]   'ROWS'
[9]   INGO3:→((ρMATRIX)[1]<I←I+1)/INGO4
[10]  TEMP←((I=ROW)/ιρROW),ι0
[11]  J←0
[12]  INGO5:→((ρTEMP)<J←J+1)/INGO3
[13]  (I;'    ',(T≠' ')/T←EXTERNAL[TEMP[J];])
[14]  →INGO5
[15]  INGO4:NR←(ρMATRIX)[1]+1
[16]  NC←(ρMATRIX)[2]
[17]  C←⍞
[18]  'MATRIX'
[19]  ⍉((NC+1),NR)ρ((¯1)+ιNR),,⍉(NR,NC)ρ(ιNC),,MATRIX
    ∇
```

```
      ∇INGMOD[⎕]∇

    ∇ INGMOD

[1]   ROW←COL←(NT←(ρEXTERNAL)[1])ρ0
[2]   NR←NC←0
[3]   COMP←0
[4]   COL[T]←ιNC←ρT←RULE[;2]
[5]   ROW[T]←ιNR←ρT←4+ιNT-4
[6]   MATRIX←(NR,NC)ρ0
[7]   I←0
[8]   TEST←ι4
[9]   INGM1:→((ρRULE)[1]<I←I+1)/INGM6
[10]  J←3
[11]  INGM2:→(RULE[I;J←J+1]=0)/INGM1
[12]  →(RULE[I;J]∊TEST)/INGM2
[13]  MATRIX[ROW[RULE[I;J]];COL[RULE[I;2]]]←1
[14]  →INGM2
[15]  C←'COMPRESS IDENTICAL ROWS'
[16]  INGM6:I←0
[17]  INGM7:→(NR≤I←I+1)/INGM8
[18]  J←I
[19]  INGM9:→(NR<J←J+1)/INGM7
[20]  →(∨/MATRIX[I;]≠MATRIX[J;])/INGM9
[21]  ROW[(J=ROW)/ιρROW]←I
[22]  →INGM9
[23]  INGM8:I←0
[24]  T←⌈/ROW
[25]  TEMP←ι0
[26]  INGM10:J←(((S←⌊/(ROW>0)/ROW)=ROW)/ιρROW),ι0
[27]  TEMP←TEMP,MATRIX[ROW[J[1]];]
[28]  ROW[J]←I←I-1
[29]  →(S≠T)/INGM10
[30]  NR←(-I)
[31]  ROW←(-ROW)
[32]  MATRIX←(NR,NC)ρTEMP
[33]  →(COMP=1)/0
[34]  C←'FILL IN CARRY OVERS'
[35]  J←0
[36]  INGM12:→(NC<J←J+1)/INGM11
[37]  I←0
[38]  INGM13:→(NR<I←I+1)/INGM12
[39]  →(~MATRIX[I;J])/INGM13
[40]  TEMP←ROW[COLιJ]
[41]  →(TEMP=I)/INGM13
[42]  MATRIX[I;]←MATRIX[I;]∨MATRIX[TEMP;]
[43]  →INGM13
[44]  INGM11:TERMA←(~(ιNT)∊(ι4),RULE[;2])/ιNT
[45]  TEMP←(ιNR)∊ROW[TERMA]
[46]  MATRIX←TEMP/[1]MATRIX
[47]  ROW←(TEMP/(ιNR))ιROW
[48]  NR←(ρMATRIX)[1]
[49]  ROW[(~(ιNT)∊TERMA)/ιNT]←0
[50]  COMP←1
[51]  →INGM6
    ∇
```

TRANSITION DIAGRAM ROUTINES

TABLE DIAGRAM PROG

This algorithm determines the structure of the input string *PROG* by using the diagrams supplied in *TABLE*.

TABLE is an $N \times 6$ element matrix where N is the number of lines in the transition diagrams. The columns of *TABLE* contain the following information:

1. The numerical representation of the syntactic unit being defined.
2. The initial node of the i^{th} connecting line.
3. The end node of the i^{th} connecting line.
4. $= 1$ - i^{th} line represents syntactic unit.
   $= 0$ i^{th} line represents terminal character.
5. > 0 numerical representation of syntactic unit or terminal character.
   $= 0$ open path.
6. $= 0$ non-exit node.
   $= 1$ exit node.
   < 0 multiple exit nodes.

CMP DIAGALTERNATE INITNODE

A recursive function which prepares the diagrams for *DIAGRAMAUTO*. The recursive feature is used when two or more definitions exist for one syntactic unit.

TABLE ← DIAGRAMAUTO

This function prepares a transition diagram for each rule from a set of syntax specifications in Irons' Notation.

TABLEO ← DIAGRAMCOR TABLE

This function is used to replace or insert rows into *TABLE*.

TABLEO ← DIAGRAMMOD TABLE

This algorithm takes a *TABLE* prepared by *DIAGRAMREAD* or *DIAGRAMAUTO* and crossreferences the individual diagrams as well as converting the linkages in the individual diagrams to linkages in terms of the entire set of diagrams.

DIAGRAMOUT TABLE

This function lists the transition diagrams contained in *TABLE*.

TABLE ← DIAGRAMREAD

This routine will read and store the numeric representation of transition diagrams which have been prepared manually.

```
∇DIAGALTERNATE[⎕]∇

    ∇ CMP DIAGALTERNATE INITNODE

[1]    →(2∊TEMP←RULE[IRL;(CMP<TEMP)/TEMP←⍳(⍴RULE)[2]])/DIAGL1
[2]    DIAGL2:→(RULE[IRL;CMP]∊ 0 2)/DIAGL8
[3]    →(RULE[IRL;CMP]=4)/DIAGL5
[4]    INODE←INITNODE
[5]    FNODE←NODE+1
[6]    →(RULE[IRL;CMP]≠3)/DIAGL4
[7]    LOOP←NODE
[8]    →DIAGL5
[9]    DIAGL4:→(RULE[IRL;CMP+1]≠4)/DIAGL6
[10]   NODE←LOOP-1
[11]   FNODE←LOOP
[12]   DIAGL6:NAME←RULE[IRL;CMP]
[13]   SYNUIND←NAME∊RULE[;2]
[14]   EXIT←0
[15]   →(~RULE[IRL;CMP+1]∊ 0 2)/DIAGL7
[16]   EXIT←1
[17]   DIAGL7:LIST←LIST,SYNU,INODE,FNODE,SYNUIND,NAME,EXIT
[18]   INITNODE←NODE←NODE+1
[19]   DIAGL5:CMP←CMP+1
[20]   →DIAGL2
[21]   DIAGL1:CMPT←1+CMP+TEMP⍳2
[22]   CMPT DIAGALTERNATE INITNODE
[23]   →DIAGL2
[24]   DIAGL8:→(LIST[⍴LIST]=1)/0
[25]   LIST←LIST,SYNU,LOOP,(LOOP+1), 0 0 1
    ∇


    ∇DIAGRAM[⎕]∇

    ∇ TABLE DIAGRAM PROG;PT

[1]    C←'INITIALIZE STACK AND CONVERT PROGRAM'
[2]    PROGT←IDENTPROGRAM PROG,' ? .'
[3]    STACK←23,⍳0
[4]    FLAG←PT←0
[5]    C←'ADVANCE POINTER FOR NEW SYMBOL'
[6]    DIAGST:INSYM←PROGT[PT←PT+1]
[7]    DIAGSU:STCKTOP←VALUE STACK
[8]    C←'CHECK FOR SYMBOL DETERMINATION'
[9]    →(TABLE[STCKTOP;4]=0)/DIAGSY
[10]   C←'ADD NEW UNIT TO STACK'
[11]   STACK←STACK,TABLE[STCKTOP;5]
[12]   →DIAGSU
[13]   C←'CHECK FOR OPEN PATH'
[14]   DIAGSY:→(TABLE[STCKTOP;5]=0)/DIAGTRE
[15]   C←'CHECK FOR SYMBOL MATCH'
```

```
[16]  →(INSYM=TABLE[STCKTOP;5])/DIAGTRS
[17]  C←'CHECK FOR ALTERNATE PATH'
[18]  DIAGNN:TEMP←(TABLE[STCKTOP;1]=TABLE[;1])/⍳(⍴TABLE)[1
      ]
[19]  ALTN←TABLE[(TEMP←(STCKTOP<TEMP)/TEMP);2]⍳TABLE[
      STCKTOP;2]
[20]  →(ALTN>⍴TEMP)/DIAGFL
[21]  STACK[⍴STACK]←TEMP[ALTN]
[22]  →DIAGSU
[23]  C←'UNIT PATH TRAVERSED'
[24]  DIAGTRU:FLAG←0
[25]  →DIAGCHK
[26]  C←'EMPTY PATH TRAVERSED'
[27]  DIAGTRE:FLAG←0
[28]  →DIAGCHK
[29]  C←'SYMBOL PATH TRAVERSED'
[30]  DIAGTRS:((1+⍴STACK);' ',((T≠' ')/T←CODE[PT;]),' TERM
      INAL')
[31]  FLAG←1
[32]  C←'CHECK FOR DIAGRAM EXIT'
[33]  DIAGCHK:→(TABLE[STCKTOP;6]≠0)/DIAGEX
[34]  STACK[⍴STACK]←TABLE[STCKTOP;3]
[35]  C←'DETERMINE RESTART POINT'
[36]  →(FLAG,~FLAG)/DIAGST,DIAGSU
[37]  C←'DIAGRAM HAS BEEN TRAVERSED'
[38]  DIAGEX:((⍴STACK);' ',(T≠' ')/T←,EXTERNAL[TABLE[
      STCKTOP;1];])
[39]  C←'CHECK FOR MULTIPLE EXITS'
[40]  →(TABLE[STCKTOP;6]<0)/DIAGMP
[41]  DIAGUS:STACK←UNSTACK STACK
[42]  INSYM←PROGT[PT←PT+FLAG]
[43]  C←'CHECK FOR PROGRAM OR ERROR'
[44]  →(((⍴STACK)=0)∧PT=⍴PROGT)/DIAGPR
[45]  →((⍴STACK)=0)/DIAGERR
[46]  STCKTOP←VALUE STACK
[47]  →DIAGTRU
[48]  C←'RESET STACK BECAUSE OF MULTI EXITS'
[49]  DIAGMP:STACK[¯1+⍴STACK]←STACK[¯1+⍴STACK]+(¯1)+|TABLE
      [STCKTOP;6]
[50]  →DIAGUS
[51]  C←'NO ALTERNATE PATHS'
[52]  DIAGFL:STACK←UNSTACK STACK
[53]  →((⍴STACK)=0)/DIAGERR
[54]  STCKTOP←VALUE STACK
[55]  →DIAGNN
[56]  C←'PROGRAM IS SYNTACTICALLY CORRECT'
[57]  DIAGPR:→0
[58]  C←'PROGRAM HAS ERROR'
[59]  DIAGERR:'ERROR'
[60]  PROG
[61]  (((PT+1)⍴' '),'∧')
[62]  →0
      ∇
```

```
      ∇DIAGRAMAUTO[⎕]∇

    ∇ TABLE←DIAGRAMAUTO

[1]   TABLE←⍳0
[2]   IRL←0
[3]   DIAGA1:→((⍴RULE)[1]<IRL←IRL+1)/DIAGA2
[4]   SYNU←RULE[IRL;2]
[5]   CMP←4
[6]   INITNODE←NODE←1
[7]   LIST←⍳0
[8]   CMP DIAGALTERNATE NODE
[9]   TABLE←TABLE,LIST
[10]  →DIAGA1
[11]  DIAGA2:TABLE←(((⍴TABLE)÷6),6)⍴TABLE
    ∇


      ∇DIAGRAMCORR[⎕]∇

    ∇ TABLEO←DIAGRAMCORR TABLE;ROW;CORR;ROWA;ROWB;TAB;NT

[1]   DIAC2:ROW←⎕
[2]   →(0=ROW)/DIAC3
[3]   CORR← [
[4]   →((⌊ROW)<ROW)/DIAC1
[5]   TABLE[ROW;]←CORR
[6]   →DIAC2
[7]   DIAC1:ROWB←(⍴TABLE)[1]-ROWA←⌊ROW
[8]   NT←⍴TAB←,TABLE
[9]   TABLE←(((⍴TAB)÷6),6)⍴TAB←((NT⍺6×ROWA)/TAB),CORR,(NT⍵
      6×ROWB)/TAB
[10]  →DIAC2
[11]  DIAC3:TABLEO←TABLE
    ∇

      ∇DIAGRAMMOD[⎕]∇

    ∇ TABLEO←DIAGRAMMOD TABLE;I

[1]   TEMP←TABLE[;4]\TABLE[;1]⍳TABLE[;4]/TABLE[;5]
[2]   TABLE[;5]←((~TABLE[;4])×TABLE[;5])+TEMP
[3]   I←0
[4]   DIAMD1:→((⍴TABLE)[1]≤I←I+1)/DIAMD2
[5]   →(TABLE[I;6]≠0)/DIAMD1
[6]   TEMP←((TABLE[I;3]=TABLE[;2])∧TABLE[I;1]=TABLE[;1])
[7]   TABLE[I;3]←(TEMP/⍳(⍴TABLE)[1])[1]
[8]   →DIAMD1
[9]   DIAMD2:TABLEO←TABLE
    ∇
```

```
      ∇DIAGRAMOUT[⎕]∇

    ∇ DIAGRAMOUT TABLE;T

[1]   ⍉(7,T)⍴(⍳T←(⍴TABLE)[1]),,⍉TABLE
    ∇


      ∇DIAGRAMREAD[⎕]∇

    ∇ TABLE←DIAGRAMREAD;LIST

[1]   TABLE←⍳0
[2]   LIST←5⍴0
[3]   DIARD1:LIST←⎕
[4]   →((LIST,⍳0)[1]=¯999)/DIARD2
[5]   TABLE←TABLE,LIST
[6]   →DIARD1
[7]   DIARD2:TABLE←(((⍴TABLE)÷6),6)⍴TABLE
    ∇
```

MULTIPLE PARSE ROUTINES

CHAINMATRIX

This algorithm prepares vectors which indicate the syntactic units that are the initial components in the definitions of other syntactic units. For each definition

DEFIN - contains the numeric representation of the unit being defined.

INITL - contains the numeric representation of the initial term in the definition.

ROWR - indicates the row index of *RULES* for this definition.

COLR - indicates the column index of *RULES* for the initial element in this definition.

I COPYPARSE J

This function copies row *I* of *PNAME* for columns 1 to *J* into the first free row of *PNAME*. Similarily for *PSYNP*. The syntax pointer in *PSYNP* is also reset.

OUTPUTPARSE I

Displays the I^{th} rows of *PNAME* and *PSYNP*.

MULTICHAIN

This routine prepares the vectors *SNAME* and *SSUCC* and the arrays *CNAME* and *CSUCC*. The syntactic specifications are contained in the numeric array *RULES*.

CNAME - indicates the chain of initial constituents of definitions.

CSUCC - points to an element in *SNAME* which must follow in order to complete this definition.

SNAME - indicates elements which must follow other elements.

$SSUCC_i$ - points to an element in *SNAME* which must follow $SNAME_i$.

= 0 No following element possible.

SYNTAX MULTIPARSE PROG

This algorithm determines all possible structures for the string of symbols contained in *PROG*. The main arrays are

PNAME - gives the numerical representation of each parse.

PSYNP - indicates the syntactic unit which must follow the corresponding element in *PNAME* in order to extend that particular parse. The element of largest index is called the syntax pointer.

IN SPLITCHAIN NEXT

This recursive function processes syntactic units which can lead to two or more syntactic units.

OUTPUTCHAIN

This function displays the vector *INITL*, *DEFIN*, *ROWR*, and *COLR*.

```
        ∇CHAINMATRIX[⎕]∇

      ∇ CHAINMATRIX

[1]     INITL←DEFIN←ROWR←COLR←⍳0
[2]     NRLS←(⍴RULES)[1]
[3]     I←0
[4]     CHAN1:→(NRLS<I←I+1)/CHAN2
[5]     J←3
[6]     CHAN3:→(RULES[I;J←J+1]=0)/CHAN1
[7]     →(RULES[I;J-1]>2)/CHAN3
[8]     INITL←INITL,RULES[I;J]
[9]     DEFIN←DEFIN,RULES[I;2]
[10]    ROWR←ROWR,I
[11]    COLR←COLR,J
[12]    →CHAN3
[13]    CHAN2:'FINI'
      ∇


        ∇COPYPARSE[⎕]∇

      ∇ I COPYPARSE J

[1]     C←'COPY PARSE IPR FROM OUTSIDE BRACKET TO BRACKET JB
        R INTO NEXT POSITION'
[2]     U←0
[3]     COPY2:→(J<U←U+1)/COPY1
[4]     PNAME[NT;U]←PNAME[I;U]
[5]     PSYNP[NT;U]←PSYNP[I;U]
[6]     →COPY2
[7]     C←'RESET SYNTAX POINTER FOR INSIDE BRACKET'
[8]     COPY1:PSYNP[NT;J]←SSUCC[|PSYNP[NT;J]]
[9]     PARSE[NT;2]←PARSE[I;2]
[10]    →0
      ∇


        ∇OUTPUTCHAIN[⎕]∇

      ∇ OUTPUTCHAIN

[1]     '      INIT DEFN ROW COL'
[2]     ⍉(5,⍴INITL)⍴(⍳⍴INITL),INITL,DEFIN,ROWR,COLR
      ∇
```

```
      ∇MULTICHAIN[⎕]∇

    ∇ MULTICHAIN

[1]   C←'INITIAL TERMINAL CHARACTERS'
[2]   CHNRL←(ρRULES)ρ0
[3]   NUMIT←ιρINITL
[4]   NRL←0
[5]   INITERM←(~INITLϵDEFIN)/NUMIT
[6]   CNAME←CSUCC← 1 0 ρ0
[7]   SNAME←SSUCC←ι0
[8]   NUMC←ι0
[9]   K←0
[10]  KCH←0
[11]  MULC1:→((ρINITERM)<K←K+1)/MULC2
[12]  KCH←KCH+1
[13]  CNAME←(TEMP←KCHαKCH-1)\CNAME
[14]  CSUCC←TEMP\CSUCC
[15]  NUMC←NUMC,0
[16]  CNAME[1;KCH]←INITL[INITERM[K]]
[17]  CSUCC[1;KCH]←0
[18]  LEAD←INITERM[K]
[19]  1 SPLITCHAIN LEAD
[20]  →MULC1
[21]  MULC2:'FINI'
    ∇



      ∇OUTPUTPARSE[⎕]∇

    ∇ OUTPUTPARSE I

[1]   →(~OUTPUT)/0
[2]   ' '
[3]   ('PARSE ';I)
[4]   TEMP←(0≠,PNAME[I;])/ι(ρPNAME)[2]
[5]   ,PNAME[I;TEMP]
[6]   ,PSYNP[I;TEMP]
    ∇
```

```
      ∇MULTIPARSE[⎕]∇

    ∇ SYNTAX MULTIPARSE PROG

[1]   OUTPUT←0
[2]   PROGT←IDENTPROGRAM PROG,' '
[3]   NCHAIN←(ρCNAME)[2]
[4]   C←'SET UP INITIAL PARSES'
[5]   PT←1
[6]   INP←(((CHAR←PROGT[PT])=CNAME[1;])/⍳NCHAIN),⍳0
[7]   NPR←ρINP
[8]   PNAME←PSYNP←(NPR,(⌈/NUMC[INP]))ρ0
[9]   NPARSE←NPRρ0
[10]  PARSE←⍉(2,NPR)ρ(NPRρ0),⍳NPR
[11]  I←0
[12]  MLTP1:→(NPR<I←I+1)/MLTP2
[13]  NPARSE[I]←NUMC[INP[I]]
[14]  PNAME[I;S←(T+1)-⍳T]←CNAME[(⍳T←NPARSE[I]);INP[I]]
[15]  PSYNP[I;S]←,CSUCC[⍳T;INP[I]]
[16]  OUTPUTPARSE I
[17]  (PARSE[I;];' * ';PNAME[I;⍳NPARSE[I]])
[18]  →MLTP1
[19]  MLTP2:NT←NPR+1
[20]  →((ρPROGT)<PT←PT+1)/0
[21]  INP←(((CHAR←PROGT[PT])=CNAME[1;])/⍳NCHAIN),⍳0
[22]  PCHAIN←ρINP
[23]  C←'CONSIDER EACH POSSIBLE PARSE'
[24]  IPR←0
[25]  MLTP3:→(NPR<IPR←IPR+1)/MLTP4
[26]  →(NT≤(ρPNAME)[1])/MLTP6
[27]  TEMPA←NTα(ρPNAME)[1]
[28]  PNAME←TEMPA\[1]PNAME
[29]  PSYNP←TEMPA\[1]PSYNP
[30]  NPARSE←TEMPA\NPARSE
[31]  PARSE←TEMPA\[1]PARSE
[32]  C←'CONSIDER EACH BRACKET FROM THE INNERMOST'
[33]  MLTP6:JBR←NPARSE[IPR]
[34]  MLTP5:→(1>JBR←JBR-1)/MLTP3
[35]  C←'GET LINK FOR THIS PARSE-BRACKET'
[36]  LINK←PSYNP[IPR;JBR]
[37]  →(LINK=0)/MLTP5
[38]  C←'BRANCH IF DIRECT MATCH'
[39]  →(SNAME[|LINK]=CHAR)/MLTP11
[40]  →MLTP7
[41]  MLTP11:IPR COPYPARSE JBR
[42]  PNAME[NT;JBR+1]←CHAR
[43]  PSYNP[NT;JBR+1]←1
[44]  NPARSE[NT]←+/0≠PNAME[NT;]
[45]  OUTPUTPARSE NT
```

```
[46]  NT←NT+1
[47]  →MLTP5
[48]  C←'CHECK FOR POSSIBLE INDIRECT EXTENSION'
[49]  MLTP7:KCH←0
[50]  MLTP8:→(PCHAIN<KCH←KCH+1)/MLTP5
[51]  →(~SNAME[|LINK]∊CNAME[;INP[KCH]])/MLTP8
[52]  IPR COPYPARSE JBR
[53]  TEMP←JBR
[54]  EXT←CNAME[;INP[KCH]]⍳SNAME[|LINK]
[55]  MLTP9:TEMP←TEMP+1
[56]  →(TEMP≤(⍴PNAME)[2])/MLTP30
[57]  TEMPA←TEMPα(⍴PNAME)[2]
[58]  PNAME←TEMPA\PNAME
[59]  PSYNP←TEMPA\PSYNP
[60]  MLTP30:PNAME[NT;TEMP]←CNAME[EXT;INP[KCH]]
[61]  PSYNP[NT;TEMP]←CSUCC[EXT;INP[KCH]]
[62]  →(EXT=1)/MLTP10
[63]  EXT←EXT-1
[64]  →MLTP9
[65]  MLTP10:NPARSE[NT]←+/0≠PNAME[NT;]
[66]  OUTPUTPARSE NT
[67]  NT←NT+1
[68]  →(NT≤(⍴PNAME)[1])/MLTP8
[69]  TEMPA←NTα(⍴PNAME)[1]
[70]  PNAME←TEMPA\[1]PNAME
[71]  PSYNP←TEMPA\[1]PSYNP
[72]  NPARSE←TEMPA\NPARSE
[73]  PARSE←TEMPA\[1]PARSE
[74]  →MLTP8
[75]  C←'ADJUST PARSE TABLE'
[76]  MLTP4:TEMP←⍳NPR
[77]  PNAME[TEMP;]←PSYNP[TEMP;]←0
[78]  PARSE[TEMP;]←0
[79]  NPARSE[TEMP]←0
[80]  NT←NT-1
[81]  NNP←NT-NPR
[82]  →(NNP=0)/MLTP40
[83]  I←0
[84]  ' '
[85]  L←0
[86]  MLTP20:→(NNP<I←I+1)/MLTP25
[87]  TEMP←I+NPR
[88]  →(∧/PNAME[TEMP;]=0)/MLTP20
[89]  L←L+1
[90]  NPARSE[L]←NPARSE[TEMP]
[91]  PARSE[L;1]←PARSE[TEMP;2]
[92]  PARSE[L;2]←L
[93]  PNAME[L;]←PNAME[TEMP;]
```

```
[94]  PSYNP[L;]←PSYNP[TEMP;]
[95]  (PARSE[L;];' * ';PNAME[L;⍳NPARSE[L]])
[96]  J←TEMP-1
[97]  MLTP21:→(NT<J←J+1)/MLTP20
[98]  →(~∧/((,PNAME[L;])=,PNAME[J;]),(,PSYNP[L;])=,PSYNP[J
      ;])/MLTP21
[99]  PNAME[J;]←PSYNP[J;]←0
[100] PARSE[J;]←0
[101] NPARSE[J]←0
[102] →MLTP21
[103] MLTP25:NPR←L
[104] →MLTP2
[105] MLTP40:('ERROR ';PT)
      ∇


      ∇SPLITCHAIN[⎕]∇

    ∇ IN SPLITCHAIN NEXT;J

[1]   NEXT←NEXT,⍳0
[2]   →(0=⍴NEXT)/0
[3]   IN←IN+1
[4]   →(IN<(⍴CNAME)[1])/SPLC4
[5]   CNAME←(TEMP←IN⍺(⍴CNAME)[1])\[1]CNAME
[6]   CSUCC←TEMP\[1]CSUCC
[7]   SPLC4:J←0
[8]   SPLC1:→((⍴NEXT)<J←J+1)/0
[9]   →(J=1)/SPLC3
[10]  IN←STRIN
[11]  KCH←KCH+1
[12]  CNAME←(TEMP←KCH⍺KCH-1)\CNAME
[13]  CSUCC←TEMP\CSUCC
[14]  NUMC←NUMC,0
[15]  CNAME[TEMP;KCH]←CNAME[(TEMP←⍳IN-1);KCH-1]
[16]  CSUCC[TEMP;KCH]←CSUCC[TEMP;KCH-1]
[17]  SPLC3:CNAME[IN;KCH]←DEFIN[NEXT[J]]
[18]  IROW←ROWR[NEXT[J]]
[19]  ICOL←1+COLR[NEXT[J]]
[20]  →((TEMP←CHNRL[IROW;ICOL])=0)/SPLC15
[21]  CSUCC[IN;KCH]←TEMP
[22]  →SPLC7
[23]  SPLC15:TEMP←RULES[IROW;ICOL]
[24]  →(TEMP∊ 0 2)/SPLC5
[25]  →(TEMP=3)/SPLC6
[26]  CSUCC[IN;KCH]←NRL←NRL+1
[27]  CHNRL[IROW;ICOL]←NRL
[28]  SNAME←SNAME,TEMP
[29]  →SPLC12
[30]  SPLC5:CSUCC [IN;KCH]←0
```

```
[31]  →SPLC7
[32]  SPLC6:CSUCC[IN;KCH]←-NRL←NRL+1
[33]  CHNRL[IROW;ICOL]←-NRL
[34]  TEMP←RULES[IROW;ICOL←ICOL+1]
[35]  SNAME←SNAME,TEMP
[36]  BRKET←NRL
[37]  SPLC8:→((TEMP←RULES[IROW;ICOL←ICOL+1])=4)/SPLC9
[38]  SSUCC←SSUCC,NRL←NRL+1
[39]  SNAME←SNAME,TEMP
[40]  →SPLC8
[41]  SPLC9:SSUCC←SSUCC,-BRKET
[42]  SPLC12:TEMP←RULES[IROW;ICOL←ICOL+1]
[43]  →(TEMP∊ 0 2)/SPLC20
[44]  →(TEMP=3)/SPLC11
[45]  SSUCC←SSUCC,NRL←NRL+1
[46]  SNAME←SNAME,TEMP
[47]  →SPLC12
[48]  SPLC11:BRKET←ICOL+1
[49]  SPLC14:→((TEMP←RULES[IROW;ICOL←ICOL+1])=4)/SPLC13
[50]  SSUCC←SSUCC,NRL←NRL+1
[51]  SNAME←SNAME,TEMP
[52]  →SPLC14
[53]  SPLC13:SSUCC←SSUCC,-BRKET
[54]  →SPLC12
[55]  SPLC20:→((⍴SNAME)=⍴SSUCC)/SPLC7
[56]  SSUCC←SSUCC,0
[57]  SPLC7:NEST←((CNAME[IN;KCH]=INITL)/NUMIT),⍳0
[58]  →(0<⍴NEST)/SPLC2
[59]  NUMC[KCH]←IN
[60]  SPLC2:IN SPLITCHAIN NEST
[61]  STRIN←IN
[62]  →SPLC1
      ∇
```

UTILITY ROUTINES

IDENTBASE

Initializes the structures *EXTERNAL*, *PRIMARY*, *AND* *OVERFLOW* and inserts the symbols ::=, |, :[, and]: into these tables.

NUMBER ← IDENTCOMP WORD

Converts the six alphanumeric characters in the vector *WORD* to a single code number, *NUMBER*, in base 200.

TEXTA ← IDENTCORR TEXT

Provides the means to correct the alphabetic vector *TEXT* which has been read by *IDENTTEXT*. The corrected text is stored in *TEXTA*.

IDENTEXT LIST

Calculates the identification numbers for and classifies n six-character alphabetic words contained in the $n \times 6$ matrix *LIST*. The results are stored in *EXTERNAL*, *PRIMARY*, and *OVERFLOW*.

NUMCODE ← IDENTFIND WORD

Determines the identification numbers *NUMCODE* for n six-character alphanumeric words contained in the $n \times 6$ matrix *WORD*. The numbers depend on the words in the matrix *EXTERNAL*. If a word is not in *EXTERNAL*, it is added.

VALUE ← IDENTIFY WORD

Determines the identification number *VALUE* for the six alphabetic characters in the vector *WORD*. The same procedure is used as in *IDENTFIND*.

LINE ← IDENTLINE SYNTEST

Provides successive parts of the alphabetic vector *SYNTEST* with each call. The parts are separated by the carriage return symbol and are returned in the vector *LINE*. *RESTART* must be set to 0 for the first call.

IDENTOUTPUT TEXT

Prints the vector *TEXT* stored by *IDENTTEXT* with the line number and the cumulative number of characters to that line.

NUMBER ← IDENTPROGRAM PROG

Converts a string of words, separated by blanks, contained in the vector *PROG* to numbers and returns the results in *NUMBER*. If a symbol is not contained in *EXTERNAL*, a number determined by *IDENTSYMBOLS* is used.

IDENTSYMBOLS

Assigns numberic codes to the terminal characters contained in *INPSYM* so as to specify the same code for equivalent symbols. The codes are contained in *INPCODE*.

RULE ← *IDENTSYNTAX SYNTAX*

Converts the alphabetic representation *SYNTAX* of a set of n rules of maximum length m to an $n \times (m+2)$ numeric matrix rule containing the identification numbers of the symbols in *SYNTAX*.

SYNTEXT ← *IDENTTEXT*

Reads and stores a vector *SYNTEXT* of alphabetic characters and separates the lines with a carriage return symbol. The routine stops when a 'Δ' is entered as the first character in a line.

CODE ← *OPERATOR OPERS*

Places a vector *OPERS* of n words separated by blanks into an $n \times 6$ matrix *CODE*. Each word is left justified in the row of *CODE*.

CODE ← *OPERATORM OPERS*

Places a vector *OPERS* of n symbols into an $n \times 6$ matrix *CODE* with the symbols in the first column.

OPEXTERNAL	Outputs *EXTERNAL* table
OPPRIMARY	Outputs *PRIMARY* table
OPOVERFLOW	Outputs *OVERFLOW* table

LOAD T

This function adds an element to one of three stacks depending on the letter supplied by *T*.

UNLOAD T

This function stores the value of the top element on one of three stacks, then removes this element from the stack.

STACKO ← UNSTACK STACK

This function produces a vector *STACKO* which is equivalent to the vector *STACK* with the last element removed.

TOP ← VALUE STACK

This function stores the value of the last element of the vector *STACK* in the variable *TOP*.

```
      ∇IDENTBASE[⎕]∇

     ∇ IDENTBASE

[1]    EXTERNAL←OVERFLOW←⍳0
[2]    PRIMARY← 200 2 ⍴0
[3]    IDENTEXT OPERATOR '::= | :[ ]: '
     ∇


       ∇IDENTCOMP[⎕]∇

     ∇ NUMBER←IDENTCOMP WORD;ALPHA;VALUE;TEMP

[1]    ALPHA←' 1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ¨¯<≤=≥>≠
       ∨∧+-×÷'
[2]    ALPHA←ALPHA,'?⍵∊⍴~↑↓⍳○*←→⍺⌈⌊_∇∆∘⎕[(])⊂⊃∩∪⊥⊤|;,.\/:'
[3]    VALUE←0,⍳¯1+⍴ALPHA
[4]    TEMP←(6⍴VALUE[ALPHA⍳WORD],6⍴0)[7-⍳6]
[5]    NUMBER←1+200|(10⊥(6⍺3)/TEMP)+10⊥(6⍵3)/TEMP
     ∇


       ∇IDENTIFY[⎕]∇

     ∇ VALUE←IDENTIFY WORD

[1]    VALUE←IDENTFIND(1 6)⍴WORD,6⍴' '
     ∇


       ∇IDENTLINE[⎕]∇

     ∇ LINE←IDENTLINE SYNTEST

[1]    LINE←⍳0
[2]    →RESTART/IDTL1
[3]    SYNTEXT←SYNTEST
[4]    RESTART←1
[5]    CR←'
       '
[6]    IDTL1:→((⍴SYNTEXT)=0)/0
[7]    TEMP←((⍴SYNTEXT)⍺SYNTEXT⍳CR)
[8]    LINE←TEMP/SYNTEXT
[9]    LINE[⍴LINE]←' '
[10]   SYNTEXT←(~TEMP)/SYNTEXT
     ∇
```

```
      ∇IDENTCORR[⎕]∇

     ∇ TEXTA←IDENTCORR TEXT

[1]    CR←'
       '
[2]    NUMBER←' 123456789'
[3]    IDTC6:LOCN←⎕,⍳0
[4]    →(LOCN[1]=0)/IDTC7
[5]    CHAR←LOCN[1]
[6]    NUMB←LOCN[2]
[7]    LINE←TEXT[(CHAR-1)+⍳NUMB]
[8]    LINE[(LINE=CR)/⍳⍴LINE]←'C̲'
[9]    LINE
[10]   CORVEC←NUMB⍴⍞,NUMB⍴' '
[11]   LINE←(TEMP←CORVEC≠'/')/LINE
[12]   CORVEC←(¯1)+NUMBER⍳TEMP/CORVEC
[13]   CORVECA←⍳0
[14]   NCOR←⍴CORVEC
[15]   I←0
[16]   IDTC3:→(NCOR<I←I+1)/IDTC1
[17]   →(CORVEC[I]≠0)/IDTC2
[18]   CORVECA←CORVECA,1
[19]   →IDTC3
[20]   IDTC2:CORVECA←CORVECA,(CORVEC[I]⍴0),1
[21]   →IDTC3
[22]   IDTC1:LINE←CORVECA\LINE
[23]   LINE
[24]   NEW←⍳0
[25]   NCOR←⍴LINE
[26]   CORR←((¯1)+CORVECA⍳0)⍴' '
[27]   CORR←NCOR⍴CORR,⍞,NCOR⍴' '
[28]   NEW←⍳0
[29]   I←0
[30]   IDTC5:→(NCOR<I←I+1)/IDTC4
[31]   NEW←NEW,1⍴((T≠' ')/T←LINE[I],CORR[I]),' '
[32]   →IDTC5
[33]   IDTC4:NEW[(NEW='C̲')/⍳⍴NEW]←CR
[34]   TEMP←(NTEXTω(NTEXT←⍴TEXT)-CHAR+NUMB-1)
[35]   TEXT←((NTEXTαCHAR-1)/TEXT),NEW,TEMP/TEXT
[36]   →IDTC6
[37]   IDTC7:TEXTA←TEXT
     ∇
```

```
      ∇IDENTEXT[⎕]∇

    ∇ IDENTEXT LIST;PT;OVFL

[1]   PT←0
[2]   IDTE1:→((ρLIST)[1]<PT←PT+1)/0
[3]   NUCODE←IDENTCOMP WORD←LIST[PT;]
[4]   →(PRIMARY[NUCODE;2]≠0)/IDTE2
[5]   PRIMARY[NUCODE;2]←1+(ρEXTERNAL)[1]
[6]   →IDTE3
[7]   IDTE2:→(∧/EXTERNAL[PRIMARY[NUCODE;2];]=WORD)/IDTE1
[8]   →(PRIMARY[NUCODE;1]≠0)/IDTE4
[9]   PRIMARY[NUCODE;1]←1+(ρOVERFLOW)[1]
[10]  →IDTE5
[11]  IDTE4:OVFL←PRIMARY[NUCODE;1]
[12]  IDTE7:→(∧/EXTERNAL[OVERFLOW[OVFL;2];]=WORD)/IDTE1
[13]  →(OVERFLOW[OVFL;1]≠0)/IDTE6
[14]  OVERFLOW[OVFL;1]←1+(ρOVERFLOW)[1]
[15]  →IDTE5
[16]  IDTE6:OVFL←OVERFLOW[OVFL;1]
[17]  →IDTE7
[18]  IDTE5:TEMP←((1+(ρOVERFLOW)[1]),2)
[19]  OVERFLOW←TEMPρ(,OVERFLOW),0,1+(ρEXTERNAL)[1]
[20]  IDTE3:TEMP←((1+(ρEXTERNAL)[1]),6)
[21]  EXTERNAL←TEMPρ(,EXTERNAL),WORD
[22]  →IDTE1
    ∇


      ∇IDENTOUTPUT[⎕]∇

    ∇ IDENTOUTPUT TEXT

[1]   RESTART←0
[2]   LINE←0
[3]   CHARS←1
[4]   IDTO1:→(0=LNG←ρTEMP←IDENTLINE TEXT)/0
[5]   ((LINE←LINE+1),CHARS;'   ',TEMP)
[6]   CHARS←CHARS+LNG
[7]   →IDTO1
    ∇
```

```
      ∇IDENTFIND[⎕]∇

    ∇ NUMCODE←IDENTFIND WORD;NUCODE;OVFL

[1]    NUMCODE←⍳0
[2]    I←0
[3]    IDTF5:→((⍴WORD)[1]<I←I+1)/0
[4]    NUCODE←IDENTCOMP TEMP←WORD[I;]
[5]    →(PRIMARY[NUCODE;2]=0)/IDTF1
[6]    →(∧/TEMP=EXTERNAL[PRIMARY[NUCODE;2];])/IDTF2
[7]    →(PRIMARY[NUCODE;1]=0)/IDTF1
[8]    OVFL←PRIMARY[NUCODE;1]
[9]    IDTF3:→(∧/TEMP=EXTERNAL[OVERFLOW[OVFL;2];])/IDTF4
[10]   →(OVERFLOW[OVFL;1]=0)/IDTF1
[11]   OVFL←OVERFLOW[OVFL;1]
[12]   →IDTF3
[13]   IDTF1:IDENTEXT(1 6)⍴TEMP
[14]   NUMCODE←NUMCODE,(⍴EXTERNAL)[1]
[15]   →IDTF5
[16]   IDTF2:NUMCODE←NUMCODE,PRIMARY[NUCODE;2]
[17]   →IDTF5
[18]   IDTF4:NUMCODE←NUMCODE,OVERFLOW[OVFL;2]
[19]   →IDTF5
    ∇


      ∇IDENTPROGRAM[⎕]∇

    ∇ NUMBER←IDENTPROGRAM PROG

[1]    NUMBER←⍳0
[2]    IDENTSYMBOLS
[3]    CODE←OPERATOR PROG
[4]    IPR←0
[5]    IDTP1:→((⍴CODE)[1]<IPR←IPR+1)/0
[6]    NUCODE←IDENTCOMP TEMP←,CODE[IPR;]
[7]    →(PRIMARY[NUCODE;2]=0)/IDTP6
[8]    →(∧/TEMP=EXTERNAL[PRIMARY[NUCODE;2];])/IDTP7
[9]    →(PRIMARY[NUCODE;1]=0)/IDTP6
[10]   OVFL←PRIMARY[NUCODE;1]
[11]   IDTP2:→(∧/TEMP=EXTERNAL[OVERFLOW[OVFL;2];])/IDTP8
[12]   →(OVERFLOW[OVFL;1]=0)/IDTP6
[13]   OVFL←OVERFLOW[OVFL;1]
[14]   →IDTP2
[15]   IDTP6:NUMBER←NUMBER,INPCODE[INPSYM⍳TEMP[1]]
[16]   →IDTP1
[17]   IDTP7:NUMBER←NUMBER,PRIMARY[NUCODE;2]
[18]   →IDTP1
[19]   IDTP8:NUMBER←NUMBER,OVERFLOW[OVFL;2]
[20]   →IDTP1
    ∇
```

```
      ∇IDENTSYMBOLS[⎕]∇

    ∇ IDENTSYMBOLS

[1]   INPSYM←'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+-×÷'
[2]   SYMBOLS←'↑∘,:;()<=>≠←⊤⊥'
[3]   INPSYM←INPSYM,SYMBOLS,'?'
[4]   INPCODE←(26ρIDENTIFY 'LETTER'),10ρIDENTIFY 'DIGIT'
[5]   INPCODE←INPCODE,(2ρIDENTIFY 'ADDOP'),2ρIDENTIFY 'MUL
      OP'
[6]   INPCODE←INPCODE,(IDENTFIND OPERATORM SYMBOLS),
      ¯99
    ∇
```

```
      ∇IDENTTEXT[⎕]∇

    ∇ SYNTEXT←IDENTTEXT

[1]   INPLIST←⍳0
[2]   SYNTEXT←⍳0
[3]   CR←'
      '
[4]   IDTT1:INPLIST←⍞
[5]   INPLIST←INPLIST,⍳0
[6]   →(INPLIST[1]='Δ')/0
[7]   SYNTEXT←SYNTEXT,INPLIST,CR
[8]   →IDTT1
    ∇
```

```
      ∇IDENTSYNTAX[⎕]∇

    ∇ RULE←IDENTSYNTAX SYNTAX

[1]   RULE←⍳0
[2]   'MAX RULE SIZE'
[3]   MAX←⎕+1
[4]   RESTART←0
[5]   IDTS1:T←IDENTLINE SYNTAX
[6]   T←OPERATOR T
[7]   →('Z'=T[1;])/0
[8]   S←⍳0
[9]   IDTS←0
[10]  IDTS2:→((⍴T)[1]<IDTS←IDTS+1)/IDTS3
[11]  S←S,IDENTFIND(1 6)⍴T[IDTS;],6⍴' '
[12]  →IDTS2
[13]  IDTS3:RULE←((1+(⍴RULE)[1]),MAX+1)⍴(,RULE),(1+⍴S),MAX
      ⍴S,MAX⍴0
[14]  →IDTS1
    ∇

      ∇OPEXTERNAL[⎕]∇

    ∇ OPEXTERNAL;I

[1]   LIMIT←⎕, 1 0
[2]   I←LIMIT[1]
[3]   STOP←(LIMIT[2]=1)/(⍴EXTERNAL)[1]
[4]   →((⍴STOP)≠0)/OPEX1
[5]   STOP←⌊/LIMIT[2],(⍴EXTERNAL)[1]
[6]   I←I-1
[7]   OPEX1:→(STOP<I←I+1)/0
[8]   (I;'  ',(EXTERNAL[I;]≠' ')/EXTERNAL[I;])
[9]   →OPEX1
    ∇

      ∇OPPRIMARY[⎕]∇

    ∇ OPPRIMARY

[1]   (PRIMARY[;2]≠0)/[1](⍉(3,T)⍴(⍳T←(⍴PRIMARY)[1]),,⍉
      PRIMARY)
    ∇

      ∇OPOVERFLOW[⎕]∇

    ∇ OPOVERFLOW

[1]   ⍉(3,T)⍴(⍳T←(⍴OVERFLOW)[1]),,⍉OVERFLOW
    ∇
```

```
      ∇OPERATOR[⎕]∇

    ∇ CODE←OPERATOR OPERS;PT;S

[1]   CODE←⍳0
[2]   S←⍳0
[3]   PT←0
[4]   OPR1:→(OPERS[PT←PT+1]=' ')/OPR2
[5]   S←S,OPERS[PT]
[6]   →OPR1
[7]   OPR2:CODE←CODE,6ρS,6ρ' '
[8]   S←⍳0
[9]   →(PT<ρOPERS)/OPR1
[10]  CODE←(((ρCODE)÷6),6)ρCODE
    ∇
```

```
      ∇OPERATORM[⎕]∇

    ∇ CODE←OPERATORM OPERS

[1]   OPERS←,⍉(2,(ρOPERS))ρOPERS,(ρOPERS)ρ' '
[2]   CODE←⍳0
[3]   S←⍳0
[4]   PT←0
[5]   OPRM1:→(OPERS[PT←PT+1]=' ')/OPRM2
[6]   S←S,OPERS[PT]
[7]   →OPRM1
[8]   OPRM2:CODE←CODE,6ρS,6ρ' '
[9]   S←⍳0
[10]  →(PT<ρOPERS)/OPRM1
[11]  CODE←(((ρCODE)÷6),6)ρCODE
    ∇
```

```
      ∇LOAD[⎕]∇

    ∇ LOAD T

[1]   →(T='GSC')/LOAD1,LOAD2,LOAD3
[2]   LOAD1:GSTACK←GSTACK,GOAL
[3]   →0
[4]   LOAD2:SSTACK←SSTACK,SOURCE
[5]   →0
[6]   LOAD3:CSTACK←CSTACK,CHAR
    ∇


      ∇UNLOAD[⎕]∇

    ∇ UNLOAD T

[1]   →(T='GSC')/UNLD1,UNLD2,UNLD3
[2]   UNLD1:GOAL←VALUE GSTACK
[3]   GSTACK←UNSTACK GSTACK
[4]   →0
[5]   UNLD2:SOURCE←VALUE SSTACK
[6]   SSTACK←UNSTACK SSTACK
[7]   →0
[8]   UNLD3:CHAR←VALUE CSTACK
[9]   CSTACK←UNSTACK CSTACK
    ∇


      ∇UNSTACK[⎕]∇

    ∇ STACKO←UNSTACK STACK

[1]   STACKO←(~(⍴STACK)ω1)/STACK
    ∇


      ∇VALUE[⎕]∇

    ∇ TOP←VALUE STACK

[1]   TOP←STACK[⍴STACK]
    ∇
```

CALLING FUNCTION / CALLED FUNCTIONS	ANALYZE	DIAGALTERNATE	DIAGRAM	DIAGRAMAUTO	IDENTBASE	IDENTEXT	IDENTFIND	IDENTIFY	IDENTOUTPUT	IDENTPROGRAM	IDENTSYMBOLS	IDENTSYNTAX	MULTICHAIN	MULTIPARSE	SPLITCHAIN	UNLOAD
COPYPARSE														1		
DIAGALTERNATE		1		1												
IDENTCOMP	1		1		1	1	1	1		1	1	1		1		
IDENTEXT					1		1	1				1				
IDENTFIND	1		1					1		1	1	1		1		
IDENTIFY	1		1							1	1			1		
IDENTLINE									1			1				
IDENTPROGRAM	1		1											1		
IDENTSYMBOLS	1		1							1				1		
LOAD	1															
OPERATOR	1		1		1					1		1		1		
OPERATORM	1		1							1	1			1		
OUTPUT PARSE														1		
SPLITCHAIN													1		1	
UNLOAD	1															
UNSTACK			1													1
VALUE			1													1

FIGURE 8

FUNCTION DEPENDENCIES

NUMERIC REPRESENTATION STRUCTURES

The three matrices used to represent the alphabetic descriptions of terms by numbers are:

EXTERNAL - A six column matrix containing the alphabetic representation of the words. The identification number assigned to a word is the row index of the word in *EXTERNAL*.

PRIMARY - A 200 × 2 element matrix. The row index is the number assigned to a word by *IDENTCOMP*. If Column 1 is nonzero it is the row index of Overflow where a word with the same code number is stored. Column 2 contains the row index of *EXTERNAL* where a word with this code number is stored.

OVERFLOW - A 2 column matrix which contains references to words which have the same number assigned to them by *IDENTCOMP*. If Column 1 is nonzero it is a row of *OVERFLOW* which references another word with the same number assigned by *IDENTCOMP*. Column 2 points to a row of *EXTERNAL* which has a word with the given code number.

APPENDIX B

PROGRAMMING LANGUAGE GRAMMARS

The Backus Normal Form and Irons' Notation for a simple test grammar and a grammar of a subset of ALGOL are given. The test grammar is equivalent to that given in Chapter 4. The two changes in notation are the symbols $\langle$ and $\rangle$ are not used and the braces { } are replaced by :[ ]: . Special routines are used to process the terminal characters.

ALGOL SYNTAX (Backus Normal Form)

```
IDENTF ::= LETTER | IDENTF LETTER
NUMBER ::= DECNUM | + DECNUM | - DECNUM | INTEGR
DECNUM ::= UNSINT . | DECFRC | UNSINT DECFRC
DECFRC ::= . UNSINT
INTEGR ::= UNSINT | + UNSINT | - UNSINT
UNSINT ::= DIGIT | UNSINT DIGIT
VARIAB ::= IDENTF
ARTEXP ::= SIMAEX | IFCLAS ARTEXP ELSE ARTEXP
SIMAEX ::= TERM | SIMAEX ADDOP TERM
IFCLAS ::= IF BOOEXP THEN
TERM ::= FACTOR | TERM MULOP FACTOR
FACTOR ::= PRIMRY | FACTOR ↑ PRIMRY
PRIMRY ::= NUMBER | VARIAB | ( ARTEXP )
ADDOP ::= + | -
MULOP ::= × | ÷
BOOEXP ::= SIMBOO | IFCLAS BOOEXP ELSE BOOEXP
SIMBOO ::= LOGVAL | RELATN | ( BOOEXP )
RELATN ::= SIMAEX RELAOP SIMAEX
RELAOP ::= < | = | > | ≠
LOGVAL ::= ⊤ | ⊥
PROGRM ::= BLOCK . | COMPST .
BLOCK ::= UNLBLK | LABEL : BLOCK
UNLBLK ::= BLKHD ; COMPTL
BLKHD ::= BEGIN DECLAR
SIMPTL ::= STATEM | SIMPTL ; STATEM
COMPTL ::= SIMPTL END
COMPST ::= UNLCMS | LABEL : COMPST
UNLCMS ::= BEGIN COMPTL
STATEM ::= UNCSTA | CONSTA | ITRSTA
UNCSTA ::= COMPST | BLOCK | BASSTA
BASSTA ::= UNLBST | LABEL : BASSTA
UNLBST ::= ASSSTA | GOTOST | IOSTAT
ASSSTA ::= LFTPTL ARTEXP | LFTPTL BOOEXP
LFTPTL ::= VARIAB ←
GOTOST ::= GOTO DESEXP
DESEXP ::= LABEL
LABEL ::= IDENTF
IOSTAT ::= REDSTA | WRTSTA
REDSTA ::= READ ( INLIST )
WRTSTA ::= WRITE ( INLIST )
INLIST ::= VARIAB | INLIST VARIAB
CONSTA ::= IFSTAT ELSE STATEM | LABEL : CONSTA
IFSTAT ::= IFCLAS STATEM
ITRSTA ::= FORSTA
FORSTA ::= FORCLA STATEM | LABEL : FORSTA
FORCLA ::= FOR VARIAB ← FORLST DO
FORLST ::= FOLSEL | FORLST , FOLSEL
FOLSEL ::= ARTEXP STEP ARTEXP UNTIL ARTEXP
DECLAR ::= TYPE TYPLST
TYPE ::= REAL | INTGER | BOOLEN
TYPLST ::= VARIAB | TYPLST , VARIAB
```

ALGOL SYNTAX (Iron's Notation)

```
IDENTF ::= LETTER :[ LETTER ]:
NUMBER ::= DECNUM | + DECNUM | - DECNUM | INTEGR
DECNUM ::= UNSINT . | DECFRC | UNSINT DECFRC
DECFRC ::= . UNSINT
INTEGR ::= UNSINT | + UNSINT | - UNSINT
UNSINT ::= DIGIT :[ DIGIT ]:
VARIAB ::= IDENTF
ARTEXP ::= SIMAEX | IFCLAS ARTEXP ELSE ARTEXP
SIMAEX ::= TERM :[ ADDOP TERM ]:
IFCLAS ::= IF BOOEXP THEN
TERM ::= FACTOR :[ MULOP FACTOR ]:
FACTOR ::= PRIMRY :[ ↑ PRIMRY ]:
PRIMRY ::= NUMBER | VARIAB | ( ARTEXP )
ADDOP ::= + | -
MULOP ::= × | ÷
BOOEXP ::= SIMBOO | IFCLAS BOOEXP ELSE BOOEXP
SIMBOO ::= LOGVAL | RELATN | ( BOOEXP )
RELATN ::= SIMAEX RELAOP SIMAEX
RELAOP ::= < | = | > | ≠
LOGVAL ::= ⊤ | ⊥
PROGRM ::= BLOCK . | COMPST .
BLOCK ::= UNLBLK | LABEL : BLOCK
UNLBLK ::= BLKHD ; COMPTL
BLKHD ::= BEGIN DECLAR :[ ; DECLAR ]:
SIMPTL ::= STATEM :[ ; STATEM ]:
COMPTL ::= SIMPTL END
COMPST ::= UNLCMS | LABEL : COMPST
UNLCMS ::= BEGIN COMPTL
STATEM ::= UNCSTA | CONSTA | ITRSTA
UNCSTA ::= COMPST | BLOCK | BASSTA
BASSTA ::= UNLBST | LABEL : BASSTA
UNLBST ::= ASSSTA | GOTOST | IOSTAT
ASSSTA ::= LFTPTL ARTEXP | LFTPTL BOOEXP
LFTPTL ::= VARIAB ←
GOTOST ::= GOTO DESEXP
DESEXP ::= LABEL
LABEL ::= IDENTF
IOSTAT ::= REDSTA | WRTSTA
REDSTA ::= READ ( INLIST )
WRTSTA ::= WRITE ( INLIST )
INLIST ::= VARIAB :[ VARIAB ]:
CONSTA ::= IFSTAT | IFSTAT ELSE STATEM | LABEL : CONSTA
IFSTAT ::= IFCLAS STATEM
ITRSTA ::= FORSTA
FORSTA ::= FORCLA STATEM | LABEL : FORSTA
FORCLA ::= FOR VARIAB ← FORLST DO
FORLST ::= FOLSEL :[ , FOLSEL ]:
FOLSEL ::= ARTEXP | ARTEXP STEP ARTEXP UNTIL ARTEXP
DECLAR ::= TYPE TYPLST
TYPE ::= REAL | INTGER | BOOLEN
TYPLST ::= VARIAB :[ , VARIAB ]:
```

ALGOL SYNTAX (Numeric Codes)

1	::=	46	*COMPST*
2	|	47	*UNLBLK*
3	:[	48	*LABEL*
4	]:	49	:
5	*IDENTF*	50	*BLKHD*
6	*LETTER*	51	;
7	*NUMBER*	52	*COMPTL*
8	*DECNUM*	53	*BEGIN*
9	+	54	*DECLAR*
10	-	55	*SIMPTL*
11	*INTEGR*	56	*STATEM*
12	*UNSINT*	57	*END*
13	.	58	*UNLCMS*
14	*DECFRC*	59	*UNCSTA*
15	*DIGIT*	60	*CONSTA*
16	*VARIAB*	61	*ITRSTA*
17	*ARTEXP*	62	*BASSTA*
18	*SIMAEX*	63	*UNLBST*
19	*IFCLAS*	64	*ASSSTA*
20	*ELSE*	65	*GOTOST*
21	*TERM*	66	*IOSTAT*
22	*ADDOP*	67	*LFTPTL*
23	*IF*	68	←
24	*BOOEXP*	69	*GOTO*
25	*THEN*	70	*DESEXP*
26	*FACTOR*	71	*REDSTA*
27	*MULOP*	72	*WRTSTA*
28	*PRIMRY*	73	*READ*
29	↑	74	*INLIST*
30	(	75	*WRITE*
31	)	76	*IFSTAT*
32	×	77	*FORSTA*
33	÷	78	*FORCLA*
34	*SIMBOO*	79	*FOR*
35	*LOGVAL*	80	*FORLST*
36	*RELATN*	81	*DO*
37	*RELAOP*	82	*FOLSEL*
38	<	83	,
39	=	84	*STEP*
40	>	85	*UNTIL*
41	≠	86	*TYPE*
42	⊤	87	*TYPLST*
43	⊥	88	*REAL*
44	*PROGRM*	89	*INTGER*
45	*BLOCK*	90	*BOOLEN*

TEST SYNTAX

BACKUS NORMAL FORM - PROSE

VARI ::= LETTER | VARI LETTER
INTEGR ::= DIGIT | INTEGR DIGIT
FACTOR ::= VARI | INTEGR | (AREXP)
TERM ::= FACTOR | TERM MULOP FACTOR
AREXP ::= TERM | AREXP ADDOP TERM
ASSIGN ::= VARI = AREXP
PROG ::= ASSIGN | PROG ; ASSIGN

NUMERIC

7	5	1	6	2	5	6	0	0	0	0
7	7	1	8	2	7	8	0	0	0	0
10	9	1	5	2	7	2	10	11	12	0
8	13	1	9	2	13	14	9	0	0	0
8	11	1	13	2	11	15	13	0	0	0
6	16	1	5	17	11	0	0	0	0	0
8	18	1	16	2	18	19	16	0	0	0

IRONS' NOTATION - PROSE

VARI ::= LETTER :[LETTER]:
INTEGR ::= DIGIT :[DIGIT]:
FACTOR ::= VARI | INTEGR | (AREXP)
TERM ::= FACTOR :[MULOP FACTOR]:
AREXP ::= TERM :[ADDOP TERM]:
ASSIGN ::= VARI = AREXP
PROG ::= ASSIGN :[; ASSIGN]:

NUMERIC

7	5	1	6	3	6	4	0	0	0	0
7	7	1	8	3	8	4	0	0	0	0
10	9	1	5	2	7	2	10	11	12	0
8	13	1	9	3	14	9	4	0	0	0
8	11	1	13	3	15	13	4	0	0	0
6	16	1	5	17	11	0	0	0	0	0
8	18	1	16	3	19	16	4	0	0	0

EXTERNAL

1	::=
2	|
3	:[
4	]:
5	*VARI*
6	*LETTER*
7	*INTEGR*
8	*DIGIT*
9	*FACTOR*
10	(
11	*AREXP*
12	)
13	*TERM*
14	*MULOP*
15	*ADDOP*
16	*ASSIGN*
17	=
18	*PROG*
19	;

PRIMARY

4	0	13
19	0	14
24	1	8
37	0	15
42	0	17
45	0	7
58	0	1
73	0	10
75	0	12
82	0	2
83	0	19
86	0	11
107	0	9
144	0	4
153	0	6
162	0	5
191	0	16
198	0	3

OVERFLOW

1	0	18

APPENDIX C

TEST EXAMPLES

THE CONVENTIONAL ALGORITHM

The parse of a statement is reconstructed from the level numbers and symbols printed by the conventional algorithm. The parse is displayed with the aid of an array which has a column for each terminal character and a row for each level number.

The level numbers indicate which row a symbol will appear in. Level numbers are opened by terminal characters and closed by non-terminal symbols. An open level number appears on the left of a column and a closed level number on the right of a column. A terminal symbol will appear in its column and metaresults will occupy one or more columns of a row indicated by the corresponding level number. Any symbol associated with a closed level number t is defined by the symbol(s) associated with level number $t + 1$.

The conventional algorithm may fail to choose the correct analysis of a statement immediately. Thus, it may be necessary to modify the output by removing all listings from the first up to, but excluding, the last listing of a terminal character. The modified output is converted by the following procedures.

Assume t is the last level number which has been either opened or closed - $t = 1$ initially. When a terminal character with level number n is encountered the level numbers $t \leq x \leq n$ are opened on the left of the column for that terminal character and the terminal symbol is placed in its column and row location. If a metaresult with level number t is encountered, level number $t + 1$ is closed if it refers to a terminal character, the metaresult is written in the row t and t is closed in the right of the column of the last terminal character processed. When level number 1 is closed the structure is complete.

CONVENTIONAL ARRAYS - TEST SYNTAX

COLUMNS

1	*VARI*
2	*INTEGR*
3	*FACTOR*
4	*TERM*
5	*AREXP*
6	*ASSIGN*
7	*PROG*

MATRIX

0	1	2	3	4	5	6	7
1	1	0	1	1	1	1	1
2	0	1	1	1	1	1	1
3	0	0	1	1	1	1	1
4	0	0	0	0	0	0	1
5	0	0	0	0	0	1	1

ROWS

1	*LETTER*
2	*DIGIT*
3	(
3	)
3	*MULOP*
3	*ADDOP*
4	;
5	=

SYNTAX TYPE TABLE

NUM	*TYPE*	*INDEX*	*TERM*	*LKFR*
1	*VARI*	5	0	1
2	*INTEGR*	7	0	3
3	*FACTOR*	9	0	5
4	*TERM*	13	0	10
5	*AREXP*	11	0	13
6	*ASSIGN*	16	0	16
7	*PROG*	18	0	19
8	*LETTER*	6	1	6
9	*DIGIT*	8	1	8
10	(	10	1	10
11	)	12	1	12
12	*MULOP*	14	1	14
13	*ADDOP*	15	1	15
14	=	17	1	17
15	;	19	1	19

SYNTAX STRUCTURE TABLE

INDEX	*TYCD*	*STRC*	*SUCC*	*ALTR*
1	6	1	2	0
2	6	1	2	¯1
3	8	1	4	0
4	8	1	4	¯1
5	5	1	0	6
6	7	1	0	7
7	10	0	8	0
8	11	0	9	0
9	12	1	0	0
10	9	1	11	0
11	14	0	12	¯1
12	9	1	11	0
13	13	1	14	0
14	15	0	15	¯1
15	13	1	14	0
16	5	0	17	0
17	17	0	18	0
18	11	1	0	0
19	16	1	20	0
20	19	0	21	¯1
21	16	1	20	0

```
    ANALYZE 'A = B + 1 ; C = D'

 4   A TERMINAL
 3   VARI
 3   = TERMINAL
 7   B TERMINAL
 6   VARI
 5   FACTOR
 4   TERM
 4   + TERMINAL
 7   1 TERMINAL
 6   INTEGR
 5   FACTOR
 4   TERM
 3   AREXP
 2   ASSIGN
 2   ; TERMINAL
 4   C TERMINAL
 3   VARI
 3   = TERMINAL
 7   D TERMINAL
 6   VARI
 5   FACTOR
 4   TERM
 3   AREXP
 2   ASSIGN
 1   PROG
```

```
1                              PROG                                    1

2             ASSIGN                    2  ;  2        ASSIGN          2

3 VARI 3  =  3       AREXP              3     3 VARI 3  =    3AREXP 3

4  A   4     4 TERM 4  +   4 TERM 4           4  C   4       4 TERM 4

             5FACTOR5      5FACTOR5                          5FACTOR5

             6 VARI 6      6INTEGR6                          6 VARI 6

             7  B   7      7  1   7                          7  D   7
```

```
      ANALYZE 'BEGIN A ← A + 1 END .'

5   BEGIN TERMINAL
4   BEGIN TERMINAL
11   A TERMINAL
10   IDENTF
9   LABEL
11   A TERMINAL
10   IDENTF
9   LABEL
14   A TERMINAL
13   IDENTF
12   VARIAB
12   ← TERMINAL
11   LFTPTL
18   A TERMINAL
17   IDENTF
16   VARIAB
15   PRIMRY
14   FACTOR
13   TERM
14   + TERMINAL
13   ADDOP
19   1 TERMINAL
18   UNSINT
19   1 TERMINAL
18   UNSINT
17   INTEGR
16   NUMBER
15   PRIMRY
14   FACTOR
13   TERM
12   SIMAEX
11   ARTEXP
10   ASSSTA
9   UNLBST
8   BASSTA
7   UNCSTA
6   STATEM
5   SIMPTL
5   END TERMINAL
4   COMPTL
3   UNLCMS
2   COMPST
2   . TERMINAL
1   PROGRM
```

THE TRANSITION DIAGRAM ALGORITHM

The parse of a statement produced by the transition diagram algorithm is displayed using almost the exact method employed for the conventional algorithm. The only difference results from the NO-BACKUP condition which insures that the algorithm will produce no incorrect parses and hence the output need not be modified.

TRANSITION DIAGRAM ARRAYS -

TEST SYNTAX

INITIAL

1	5	1	2	0	6	0
2	5	2	2	0	6	0
3	5	2	3	0	0	1
4	7	1	2	0	8	0
5	7	2	2	0	8	0
6	7	2	3	0	0	1
7	9	1	2	0	10	0
8	9	2	3	1	11	0
9	9	3	4	0	12	1
10	9	1	5	1	7	1
11	9	1	6	1	5	1
12	13	1	2	1	9	0
13	13	2	3	0	14	0
14	13	3	2	1	9	0
15	13	2	3	0	0	1
16	11	1	2	1	13	0
17	11	2	3	0	15	0
18	11	3	2	1	13	0
19	11	2	3	0	0	1
20	16	1	2	1	5	0
21	16	2	3	0	17	0
22	16	3	4	1	11	1
23	18	1	2	1	16	0
24	18	2	3	0	19	0
25	18	3	2	1	16	0
26	18	2	3	0	0	1

CONNECTED

1	5	1	2	0	6	0
2	5	2	2	0	6	0
3	5	2	3	0	0	1
4	7	1	5	0	8	0
5	7	2	5	0	8	0
6	7	2	3	0	0	1
7	9	1	8	0	10	0
8	9	2	9	1	16	0
9	9	3	4	0	12	1
10	9	1	5	1	4	1
11	9	1	6	1	1	1
12	13	1	13	1	7	0
13	13	2	14	0	14	0
14	13	3	13	1	7	0
15	13	2	3	0	0	1
16	11	1	17	1	12	0
17	11	2	18	0	15	0
18	11	3	17	1	12	0
19	11	2	3	0	0	1
20	16	1	21	1	1	0
21	16	2	22	0	17	0
22	16	3	4	1	16	1
23	18	1	24	1	20	0
24	18	2	25	0	19	0
25	18	3	24	1	20	0
26	18	2	3	0	0	1

```
      TABLE DIAGRAM 'A = B + 1 ; C = D'

4 A TERMINAL
3 VARI
3 = TERMINAL
7 B TERMINAL
6 VARI
5 FACTOR
4 TERM
4 + TERMINAL
7 1 TERMINAL
6 INTEGR
5 FACTOR
4 TERM
3 AREXP
2 ASSIGN
2 ; TERMINAL
4 C TERMINAL
3 VARI
3 = TERMINAL
7 D TERMINAL
6 VARI
5 FACTOR
4 TERM
3 AREXP
2 ASSIGN
1 PROG
```

THE MULTIPLE PARSE ALGORITHM

For each terminal symbol this algorithm produces vectors of numbers representing the chains of metaresults which the terminal symbol can define on the basis of the preceding elements. Two pointers are provided for each vector. The second pointer numbers the vectors for this symbol. The first pointer indicates which vector for the preceding symbol was used to generate the given vector. If the grammar is unambiguous, there will only be one vector for the last terminal symbol.

The first step in reconstructing the parse is to select the vectors determined by the last terminal symbol and the pointers provided. As for the conventional algorithm, the parse can be represented by an array. The array is completed by listing the appropriate symbols where the vector number indicates the column and the index of the element indicates the row. If a symbol appears in more than one column of a row, it should be listed only once with indicators used to mark the ends of the appropriate columns. A metaresult in row t is defined by the elements in row $t + 1$ which are between the row t indicators.

MULTIPLE PARSE ARRAYS - TEST SYNTAX

	INIT	*DEFN*	*ROW*	*COL*
1	6	5	1	4
2	8	7	2	4
3	5	9	3	4
4	7	9	3	6
5	10	9	3	8
6	9	13	4	4
7	13	11	5	4
8	5	16	6	4
9	16	18	7	4

INITERM

1 2 5

CNAME

6	6	8	10
5	5	7	9
9	16	9	13
13	18	13	11
11	0	11	0

CSUCC

0	0	0	0
¯1	¯1	¯10	11
0	6	0	¯2
¯2	¯8	¯2	¯4
¯4	0	¯4	0

NUMC

5 4 5 4

SNAME

6 14 9 15 13 17 11 19 16 8 11 12

SSUCC

¯1 3 ¯2 5 ¯4 7 0 9 ¯8 ¯10 12 0

```
      0 MULTIPARSE 'A = B + 1 ; C = D'

 0  1 * 11  13  9  5  6
 0  2 * 18  16  5  6

 2  1 * 18  16  17

 1  1 * 18  16  11  13  9  5  6

 1  1 * 18  16  11  15

 1  1 * 18  16  11  13  9  7  8

 1  1 * 18  19

 1  1 * 18  16  5  6

 1  1 * 18  16  17

 1  1 * 18  16  11  13  9  5  6




*                                PROG                                *

*                ASSIGN                  *  ;  *        ASSIGN       *

* VARI *  =   *           AREXP          *     * VARI *  =   * AREXP*

*LETTER*      * TERM *  +   * TERM *     *LETTER*      * TERM *

   A          *FACTOR*      *FACTOR*        C          *FACTOR*

              * VARI *      *INTEGR*                   * VARI *

              *LETTER*      * DIGIT*                   *LETTER*

                  B             1                          D
```

```
      0 MULTIPARSE 'BEGIN A ← B END .'

0  1 * 44  45  47  50  53
0  2 * 52  55  56  59  45  47  50  53
0  3 * 44  46  58  53
0  4 * 52  55  56  59  46  58  53

3  1 * 44  46  58  52  55  56  59  62  63  64  67  16
      5  6
3  2 * 44  46  58  52  55  56  59  62  63  64  67  16
      5  6
3  3 * 44  46  58  52  55  56  59  45  48  5  6
3  4 * 44  46  58  52  55  56  59  46  48  5  6
3  5 * 44  46  58  52  55  56  59  62  48  5  6
3  6 * 44  46  58  52  55  56  60  48  5  6
3  7 * 44  46  58  52  55  56  61  77  48  5  6
4  8 * 52  55  56  59  46  58  52  55  56  59  62  63
      64  67  16  5  6
4  9 * 52  55  56  59  46  58  52  55  56  59  62  63
      64  67  16  5  6
4  10 * 52  55  56  59  46  58  52  55  56  59  45
      48  5  6
4  11 * 52  55  56  59  46  58  52  55  56  59  46
      48  5  6
4  12 * 52  55  56  59  46  58  52  55  56  59  62
      48  5  6
4  13 * 52  55  56  59  46  58  52  55  56  60  48
      5  6
4  14 * 52  55  56  59  46  58  52  55  56  61  77
      48  5  6

1  1 * 44  46  58  52  55  56  59  62  63  64  67  68
2  2 * 44  46  58  52  55  56  59  62  63  64  67  68
8  3 * 52  55  56  59  46  58  52  55  56  59  62  63
      64  67  68
9  4 * 52  55  56  59  46  58  52  55  56  59  62  63
      64  67  68

1  1 * 44  46  58  52  55  56  59  62  63  64  17  18
      21  26  28  16  5  6
2  2 * 44  46  58  52  55  56  59  62  63  64  24  34
      36  18  21  26  28  16  5  6
3  3 * 52  55  56  59  46  58  52  55  56  59  62  63
      64  17  18  21  26  28  16  5  6
4  4 * 52  55  56  59  46  58  52  55  56  59  62  63
      64  24  34  36  18  21  26  28  16  5  6

1  1 * 44  46  58  52  57
3  2 * 52  55  56  59  46  58  52  57
3  3 * 52  57

1  1 * 44  13
```